APPLICATIONS OF LEXICOGRAPHIC BREADTH-FIRST SEARCH TO MODULAR DECOMPOSITION, SPLIT DECOMPOSITION, AND CIRCLE GRAPHS

by

Marc Tedder

A thesis submitted in conformity with the requirements for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Copyright © 2011 by Marc Tedder
This thesis presents the first sub-quadratic circle graph recognition algorithm, and develops improved algorithms for two important hierarchical decomposition schemes: modular decomposition and split decomposition. The modular decomposition algorithm results from unifying two different approaches previously employed to solve the problem: divide-and-conquer and factorizing permutations. It runs in linear-time, and is straightforward in its understanding, correctness, and implementation. It merely requires a collection of trees and simple traversals of these trees. The split-decomposition algorithm is similar in being straightforward in its understanding and correctness. An efficient implementation of the algorithm is described that uses the union-find data-structure. A novel charging argument is used to prove the running-time. The algorithm is the first to use the recent reformulation of split decomposition in terms of graph-labelled trees. This facilitates its extension to circle graph recognition. In particular, it allows us to efficiently apply a new lexicographic breadth-first search characterization of circle graphs developed in the thesis. Lexicographic breadth-first search is additionally responsible for the efficiency of the split decomposition algorithm, and contributes to the simplicity of the modular decomposition algorithm.
Acknowledgements

It is no secret to those who know me that I have had difficulty reconciling myself to my subject. What has made it easier – and made this thesis possible – is the tremendous kindness, generosity, and patience of a number of people. My supervisor, Derek Corneil, is foremost amongst these. He has somehow managed to endure my persistent irreverence toward the subject to which he has dedicated his life. However, it is not so much his dedication that has tolerated me as it has been his understanding. This has afforded me the luxury of arriving at this point on my own terms, and along the way, presented me the opportunities for my success. Although his role in this signals his recognition of its importance, he cannot possibly appreciate my gratitude. I could not have been more fortunate to have had him as my supervisor.

It is not often that one gets the chance to say that their coauthors changed their life. But that is what Michel Habib and Christophe Paul did when they invited me to study with them in France for nearly two months in 2008. As clichéd though it may be, one cannot spend a month in Paris without being changed. Of course, spending two weeks in Provence has no less effect. But what I will remember as much as the museums and the aqueducts is the enthusiasm of both Michel and Christophe. It has the quality of making you want to learn. My greatest wish would be to learn some of their skills in seeing connections between problems. And from my other coauthor, Emeric Gioan, whose introduction in Montpellier I owe to Christophe, my wish would be to learn some of his skills in seeing problems so sharply. My hope is that this thesis reflects some progress in these pursuits.

Countless people from the Department of Computer Science at the University of Toronto deserve recognition for their contributions to this thesis, both direct and indirect. My PhD Supervisory Committee recommended content changes that have improved its quality, and have been forgiving enough to permit my unorthodox road to completion. That road was partially paved by the assistance of Linda Chow in the graduate office, who never failed to come through for me. All the while, my friends in the department have helped make these the best years of my life. Too often we don’t realize this until after such years have passed. They made me realize this each day I came to the office. It seems undignified to mention each by name, but I will single out Ingrid. For it is she who introduced me to Yuki. And it is Yuki who has given all of this meaning.
# Contents

1 **Introduction** 1  
1.1 Overview of the Thesis 3

2 **Background** 4  
2.1 Modular Decomposition 4  
2.2 Split Decomposition 7  
2.3 Circle Graphs 10  
2.4 Lexicographic Breadth-First Search 12

3 **Technical Background** 15  
3.1 Preliminaries 15  
3.2 Modular Decomposition 17  
3.3 Split Decomposition 19  
3.3.1 Graph-Labelled Trees 21  
3.4 Circle Graphs 28  
3.5 Lexicographic Breadth-First Search 30

4 **Simple, Linear-Time Modular Decomposition** 33  
4.1 Dividing by LBFS 33  
4.2 Modules Not Containing the Pivot 36  
4.3 Factorizing Permutation 43  
4.4 Modules Containing the Pivot 49  
4.5 Conquering 54  
4.6 Implementation and Running Time 57  
4.6.1 Dividing by LBFS 59  
4.6.2 Modules Not Containing the Pivot 59
6.3 Incremental Circle Graph Recognition ........................................... 127
  6.3.1 Algorithm ............................................................................. 130
  6.3.2 Implementation and Running Time ......................................... 132

7 Conclusion .................................................................................. 138
  7.1 Modular Decomposition ........................................................... 138
    7.1.1 Practical Applications ......................................................... 139
    7.1.2 Transitive Orientation ......................................................... 139
    7.1.3 Comparative Analysis ........................................................ 140
  7.2 Split Decomposition ................................................................. 141
    7.2.1 Linear-Time and Union-Find .............................................. 141
    7.2.2 Linear-time Without Union-Find ........................................... 142
    7.2.3 Simpler Split-Decomposition? .............................................. 143
  7.3 Circle Graphs ......................................................................... 144
    7.3.1 Linear-Time? ................................................................. 144
    7.3.2 Linear-Time Recognition of Permutation Graphs .................. 145
    7.3.3 Graph-Labelled Trees and their Generalizations ................. 146
  7.4 Lexicographic-Breadth First Search ........................................... 146
    7.4.1 A New LBFS Paradigm ....................................................... 147
    7.4.2 Recognition of Chordal Bipartite Graphs ............................. 147

Bibliography ................................................................................. 148
List of Figures

2.1 The set $H = \{a, b, c, d\}$ is a module. ........................................... 5

2.2 Composing two graphs by adding all possible edges between $A' = \{a, b, c, d\}$ and $B' = \{e, f\}$. If $A = \{a, b, c, d, k, \ell\}$ and $B = \{e, f, g, h, i, j\}$, then $(A, B)$ is a split in the resulting graph. ................................................................. 7

2.3 A chord diagram on the left, and its corresponding circle graph on the right. .......... 10

2.4 The chords corresponding to $N(x)$ appear consecutively without repetition. .......... 12

3.1 The modular decomposition tree for the graph in figure 2.1. We use “P” to stand for prime, and 0 and 1 to stand for “parallel” and “series”, respectively. The use of 0 and 1 is common in the literature. ......................................................... 18

3.2 The split decomposition tree of the graph in figure 2.2. ................................. 21

3.3 An example of a graph-labelled tree. .................................................... 22

3.4 The GLT induced by the nodes not incident to $e$ and $f$ in the GLT of figure 3.3. ... 23

3.5 Grafting two GLTs with respect to $q$ and $q'$. ........................................ 24

3.6 An example of the node-join and node-split operations. ............................... 26

3.7 An example of the clique-join and clique-split operations. ............................ 26

3.8 An example of the star-join and star-split operations. .................................. 27

3.9 Chord diagrams for two degenerate graphs: on the left, a clique on four vertices; on the right, a star with three extremities. Chord diagrams can be obtained for any degenerate graph by following the patterns above. .......................... 30

4.1 An example graph to demonstrate the MD algorithm’s operation. ...................... 34

4.2 Recursively computed MD trees for each maximal slice in the maximal slice partition \{x\}, \{a, b, c, d\}, \{e, y, z, f, g\}, \{h, i, j, k\} of the graph in figure 4.1. ............. 40
4.3 The recursively computed MD trees of figure 4.2, after being refined according to algorithm 4.2.1. The darkly shaded nodes are those labelled *dead*. Nodes labelled *right* are those with diagonal shading up to the right, while those nodes labelled *left* are diagonally shaded up to the left. 

4.4 The refined MD trees from figure 4.3 after nodes have been labelled *zombie* and children have been reordered by algorithm 4.3.1: zombie nodes are indicated by cross-hatching; c.f. figure 4.3 to observe the reordering of children.

4.5 The trees of figure 4.4 after the *dead* and *zombie* nodes have been deleted by algorithm 4.3.1.

4.6 The spine built by algorithm 4.4.1 for the pivot factorizing permutation from equation (4.1), based on the $\mu$- and $\rho$-values from equation (4.2).

4.7 The spine from figure 4.6 with the (co-)component leaves replaced by their corresponding trees from figure 4.5, as performed by algorithm 4.5.1 after calling algorithm 4.4.1. The resulting tree is the MD tree for the graph in figure 4.1.

5.1 Forming $ST(G + (x, S))$ from $ST(G)$ by replacing an edge with a degenerate node, as required by algorithm 5.1.3. We use “P” for “perfect” and “E” for “empty”.

5.2 An example of cleaning a fully-mixed node $u$. The triangle represents a subtree of the GLT. The dashed lines delineate portions of the GLT that would be pruned as a result of newly created perfect/empty marker vertices. Labels have been assigned to certain marker vertices to indicate their state: “P” for “perfect”, “E” for “empty”, and “M” for “mixed”. The remaining labels represent leaves in the GLT.

5.3 An example of cleaning under the assumption that states are assigned with respect to the set $S = \{c, h, g, e, f, k\}$. Labels have been assigned to certain marker vertices to indicate their state: “P” for “perfect”, “E” for “empty”, and “M” for “mixed”. On the left, the delimited box represents the set of fully-mixed nodes. Notice that the degenerate node within the box is not cleaned. The effect of cleaning is demonstrated on the right, with the nodes in the delimited box now being cleaned and fully-mixed.

5.4 An example of algorithm 5.1.5 after cleaning. On the left, the cleaned, fully-mixed nodes of figure 5.3 are contracted into a single node. On the right, a new vertex $x$ adjacent to $\{c, h, g, e, f, k\}$ is added. Labels have been assigned to certain marker vertices with respect to this set of neighbours; we use “P” for “perfect”, “E” for “empty”, and “M” for “mixed”.

viii
5.5 A parent and child in the split-tree, alongside their data-structure encoding. States are represented by “p” for “perfect”, “e” for “empty”, and “m” for “mixed”. 90

5.6 Example implementation of a star-split during perfect cleaning. We use “P” to mark the marker vertices that are perfect. On the right, the labelled marker vertices are those that have been reused, while the unlabelled ones are newly created. 99

5.7 Example implementation of a child-extremity node-join during contraction. Labelled marker vertices are those that are reused. Notice the change in $T(d)$ from left to right.102

5.8 On the left, a path $G$ and its split tree $ST(G)$. On the right, the split tree for $ST(G + (x, S))$, where $S$ consists of the ends of the path. 104

6.1 Adding a vertex to a chord diagram when its neighbourhood is consecutive. 117

6.2 On the right, a chord diagram that is partitionable with respect to the split $(A, B)$ on the left. 123

6.3 Two chord diagrams that cannot be partitioned with respect to the demonstrated split. 124

7.1 An input graph that provides a lower-bound for the implementation of our split decomposition algorithm without union-find. The required LBFS ordering is produced by starting at the root, and exploring the tree by always choosing the leftmost unexplored child. Contraction would result from processing each vertex on the bottom layer of the tree. The resulting number of parent pointer updates would be $\Omega(n^{1.5})$. 143

7.2 A permutation graph on the right, and its two different interpretations on the left. The dotted lines demonstrate how the intersection model can be transformed into a chord diagram, explicitly demonstrating the connection to circle graphs. 145
## List of Algorithms

3.5.1 \(LBFS(G)\) ................................................................. 30
4.1.1 \(DivideMDTree(S, P)\) .................................................. 37
4.2.1 \(TreeRefinement(T)\) .................................................. 39
4.3.1 \(Factorize(T)\) ........................................................... 44
4.4.1 \(BuildSpine(\sigma)\) .................................................... 53
4.5.1 \(ConquerMDTree(T)\) .................................................. 56
4.5.2 \(MDTree(G)\) ............................................................. 57
5.1.1 \(Pendant(ST(G), y)\) ................................................... 64
5.1.2 \(Pruning((T, \mathcal{F}), S)\) .............................................. 66
5.1.3 \(Unmixed(ST(G), S, u, x)\) .......................................... 69
5.1.4 \(Contraction(ST(G), S, x)\) .......................................... 75
5.1.5 \(AddVertex(ST(G), x, S)\) ........................................... 78
5.2.1 \(LBFSIncrementalSplitTree(G)\) .................................. 83
5.3.1 [77] Computing the Subtree Spanning a Set of Leaves .......... 94
6.2.1 \(ConsecContractAdd(ST(G), \mathcal{R}, x)\) ............................ 122
6.3.1 \(ConsecDegRealizer(G, S, S')\) ...................................... 129
6.3.2 \(ConsecPrimeRecog(\pi, S, S')\) .................................... 130
6.3.3 \(CircleGraphRecog(G)\) ............................................... 133
Chapter 1

Introduction

Graph decomposition is one of the most important tools in understanding the structure of a graph. Many decomposition schemes are hierarchial in nature. This allows them to be viewed through the inverse lens of graph composition. That perspective explains their success in solving problems through dynamic programming and divide and conquer. The bottleneck in these and their numerous other applications is computing the decomposition. This thesis considers that problem for two important hierarchical decomposition schemes: modular decomposition [71] and split decomposition [50].

Both modular decomposition and split decomposition are related to many other hierarchical decomposition schemes. Modular decomposition has been rediscovered many times under different names. Split decomposition is actually a generalization of modular decomposition. The two of them are themselves special cases of a hierarchical decomposition theory developed by Cunningham and Edmonds [51]. A generalization of split decomposition is the 2-join decomposition, used in the proof of the Strong Perfect Graph Theorem [31]; the recently introduced rank-width [111] can be seen as another generalization. The related clique-width [46] was introduced as an extension of modular decomposition. Together with modular decomposition and split decomposition, these hierarchical decomposition schemes have helped shape graph theory over the past two decades.

Like most hierarchical decomposition schemes, modular decomposition and split decomposition can alternately be seen as ways of constructing graphs. The general process starts with certain basic, indecomposable graphs, and then a set of composition rules is recursively applied to build the graph. The construction can be represented by a tree whose nodes and edges define certain structural relationships in the underlying graph. This tree is the basis for the many applications of modular decomposition, split decomposition, and other hierarchical decomposition schemes [45].

One of the most common applications is in the design of graph algorithms. The hierar-
chical decomposition trees naturally lend themselves to dynamic programming and divide and conquer strategies. The structure encoded by nodes and edges defines how subproblems are solved/combined. Many NP-hard optimization problems are polynomially solvable on graphs whose decomposition trees have bounded “width”. This is a complexity measure of the tree: the degree of certain nodes, for example. Popular width-parameters for which this approach has been successful are tree-width, branch-width, clique-width, and rank-width (see [90]). Modular decomposition [106] and split decomposition [118] are also amenable to this kind of approach.

There are myriad algorithmic applications of modular decomposition and split decomposition beyond these. They span both theoretical and practical domains and are far too numerous to be listed here. Habib and Paul recently surveyed the topic for modular decomposition [86]. They recognize that computing the modular decomposition tree is the implicit bottleneck in any of its applications. Of course, this holds for all decomposition schemes, including split decomposition. Accordingly, significant attention has been focused on the problem (see, e.g. [7, 55, 56, 67, 109]). This can sometimes lead to other interesting results; for example, rank-width arose in the development of a clique-width decomposition algorithm [111]. This thesis presents improved algorithms for modular decomposition and split decomposition, and then shows how the split decomposition algorithm leads to faster circle graph recognition.

Linear time algorithms for modular decomposition had previously suffered either from being highly complicated and difficult to understand, or by requiring advanced data-structures for their implementation. Our modular decomposition algorithm avoids these limitations, yet still runs in linear-time. It thereby answers a question of McConnell and Spinrad concerning the existence of such an algorithm [103].

The situation for split decomposition was similar. There had been a single linear-time split decomposition algorithm [55], albeit one of considerable difficulty. That algorithm was due to Dahlhaus, and it has recently been reformulated in [27] with an improved presentation. The result is a simpler variant that also runs in linear-time. The key parts of Dahlhaus’ complicated algorithm have been elicited, simplified, and unified. Nevertheless, one problem is that, like Dahlhaus’ algorithm, it does not seem to extend to circle graph recognition as prior split decomposition algorithms had.

Our split decomposition algorithm was developed independently. Its primary advantage over Dahlhaus’ algorithm – and the recent reformulation in [27] – is that ours extends to circle graph recognition. We use *Lexicographic Breadth-First Search* (LBFS) [119] to define a characteristic ordering of vertices in a circle graph. An incremental circle graph recognition algorithm naturally arises. It can be efficiently implemented by extending our split decomposition algorithm. The
result is the first sub-quadratic circle graph recognition algorithm.

There is little tradeoff in achieving this. Our split decomposition algorithm is both simple and efficient. That is due largely to our algorithm using the recent reformulation of split decomposition in terms of Graph-labelled Trees (GLTs) [77]. We are the first to use them in computing split decomposition. Their structure facilitates an incremental characterization of split decomposition that we develop as the basis for our algorithm. We also demonstrate an efficient implementation. Its only drawback is using union-find [33], which sacrifices linear-time by introducing an inverse Ackermann function factor into the running-time. However, this function is extremely slowly growing, and is effectively constant for all meaningful values (pg. 521-522, [33]).

The efficiency of our split decomposition algorithm also depends on LBFS, which provides a unifying theme for the results in this thesis. In particular, our split decomposition implementation adds vertices according to an LBFS ordering, taking advantage of how LBFS regulates the creation of splits. We have already mentioned the role of LBFS in circle graph recognition. For modular decomposition, we define a new recursive approach to computing LBFS that is necessary for achieving the efficiency and simplicity of the algorithm. Further context is provided in the next chapter by addressing each of these topics in detail. We conclude this chapter by providing an outline of the thesis.

1.1 Overview of the Thesis

There is a chapter for each of the three main results of this thesis: simpler modular decomposition (chapter 4), simpler split decomposition (chapter 5), and faster circle graph recognition (chapter 6). Chapter 2 provides detailed background information for each of these problems. Technical details needed for the understanding of our results appear in chapter 3. We conclude in chapter 7 with a summary, and address outstanding questions related to these results, including suggestions for future research.
Chapter 2

Background

The topics of Modular Decomposition, Split Decomposition, Circle Graphs, and Lexicographic Breadth-First search from chapter 1 are revisited here. We give informal definitions of each and expand on their individual importance. This is used to motivate the problems addressed in this thesis. An overview is also provided of results that relate to our solutions of these problems. A technical treatment of these topics is delayed until the next chapter. Throughout the thesis, \( m \) and \( n \) are used to denote the number of edges and vertices in the graph, respectively. The next chapter provides the remaining preliminaries for understanding the results in this thesis.

2.1 Modular Decomposition

Modules arise from a very natural substitution operation on graphs. Consider replacing a single vertex \( x \) in a graph \( G \) by another graph \( H \) such that each vertex in \( H \) is made universal to the neighbourhood of \( x \), and the adjacencies within \( H \) are maintained. In a sense, \( H \) is being “injected” into \( x \). A module is what results. More specifically, a module is a set of vertices, each of whose neighbourhood is the same outside the module; an example appears in figure 2.1. Notice that the vertices of \( H \) form a module in the graph resulting from the substitution. Modular Decomposition recursively performs the inverse operation: identifying a module and collapsing it into a single vertex.

Many other combinatorial objects are amenable to substitution operations of this kind. Indeed, the notion of a module and the idea of modular decomposition have been rediscovered many times in different settings. There are modular decompositions for directed graphs [100], 2-structures [63], hypergraphs, and matroids. Mohring [105] surveys applications of modular decomposition for boolean functions, set systems, and relations (including graphs). While he refers to modules as
autonomous sets, they have also appeared in the literature as: closed sets [71], clans [63], clumps [6], partitive sets [81], and externally related sets [84]. Modular decomposition, on the other hand, has alternately been discovered as substitution decomposition [105], prime-tree decomposition [63], and X-join [84].

Gallai is traditionally credited with initiating the study of modular decomposition in his seminal work on transitive orientation and the structure of comparability graphs [71]. Since then, modular decomposition has become an essential tool in perfect graph theory – especially its algorithmic component (see, e.g. [16, 82]). The fastest transitive orientation algorithms are all based on modular decomposition (e.g. [102]). It is frequently employed in recognizing different classes of graphs, including interval graphs [104], permutation graphs [115], and cographs [42]. This has continued more recently, as evidenced by [25], which uses modular decomposition to recognize probe permutation graphs. An older survey of its application to combinatorial optimization is [106], while more recent examples are [96, 91, 59, 54, 8]. New applications are continually discovered. It has been used in graph drawing [113], bioinformatics [70], and clustering [117]. The survey by Habib and Paul highlights its use in pattern matching, computational biology, and parameterized complexity [86]. Modular decomposition remains one of the most important graph decomposition schemes.

Attention has continuously focused on the efficient computation of the modular decomposition tree. In the future, we will refer to this simply as computing modular decomposition. The first polynomial time algorithm, which ran in time $O(n^4)$, dates to the early 1970’s [48]. Incremental improvements followed in the years after (e.g. [20, 107, 124]). Faster algorithms were also developed for special cases and generalizations (e.g. [81, 42]). These efforts culminated in 1994 with the first linear-time modular decomposition algorithms, independently discovered by Cournier and Habib [47] and McConnell and Spinrad [101]. Both were highly complicated and viewed primarily as theoretical contributions. Spinrad himself hoped for their simplification (pg. 149, [127]). Many attempts have since been made toward that goal. Unfortunately, these have all either had to
sacrifice linear-time for their simplicity, or have required, “conceptually difficult tricks”, “careful charging arguments to prove the time bound”, and “sophisticated union-find data structures” (pg. 59, [103]).

Attempts at simplifying the linear-time computation of modular decomposition can be classified into three streams. The first is based on a technique pioneered by Ehrenfeucht et al. [63], and later improved upon by Dahlhaus [53]. The idea is to choose a pivot vertex, say $x$, and recursively compute the modular decomposition tree for its neighbourhood and its non-neighbourhood. The modules not containing $x$ can be computed from these trees. The modules containing $x$ are computed from those not containing $x$. The tree is then assembled from the two types of modules. Although conceptually simple, computing the modules containing $x$ has proved a bottleneck. In [103], they settle for an $O(n + m \log n)$ implementation. Linear-time is achieved in [56], but the quotes above refer to that algorithm.

Another approach has been to compute an implicit representation of the tree called a factorizing permutation. This is an ordering in which the vertices descend from each node appear consecutively. The idea was introduced by Capelle and Habib [23] in an attempt to avoid the bottleneck described above. Given such a permutation, they demonstrate an algorithm to compute the modular decomposition tree in linear-time [24]. Two recent papers [136, 5] established a connection between this and problems arising in computational biology. They applied techniques developed in identifying common intervals in sets of permutations to constructing the modular decomposition tree from a given factorizing permutation. The result is $O(n + m \log n)$ [5] and $O(n + m)$ [136] algorithms solving this problem. Nevertheless, the difficulty remained in computing the factorizing permutation in the first place. Although a linear-time algorithm was claimed in [83], it was later discovered to have an error [114].

Research in the area has also sought to generalize cograph recognition algorithms. Cographs are those graphs that are fully decomposable with respect to modular decomposition. The first linear-time cograph recognition algorithm is due to Corneil et al. [42]. It operates incrementally, building the modular decomposition tree one vertex at a time, stopping when the graph first ceases to be a cograph. Spinrad introduced a data-structure called $P_4$-trees whose incremental computation generalized (and employed) the algorithm of Corneil et al.. Its computation was part of a modular decomposition algorithm that ran in time $O(m \alpha(n, m))$, where $\alpha$ is the inverse of Ackermann’s function [124]. A simpler cograph recognition algorithm based on LBFS has lately been proposed in [18]. Unfortunately, attempts at adapting it to the general modular decomposition setting have so far not been able to preserve its simplicity [36].

The linear-time modular decomposition algorithm presented (chapter 4) in this thesis unifies
the three approaches above. It generalizes a cograph recognition algorithm the author developed for a course project, and LBFS does contribute to its simplicity. However, the breakthrough really comes from introducing factorizing permutations to the recursive approach of Ehrenfeucht et al. and Dahlhaus. The two are complementary: the former allows for efficient computation of the latter, while the latter removes the bottleneck of the former. The only data-structure required to accomplish this is an ordered list of trees. McConnell and Spinrad wondered in [103] if this recursive approach could eventually yield a simple linear-time modular decomposition algorithm. We are able to answer that question in the affirmative.

2.2 Split Decomposition

Cunningham and Edmonds developed a general theory of hierarchical decompositions that yield canonical – or unique – decomposition trees [51]. A special case applying to strongly connected, directed graphs was explored by Cunningham in [50] under the name join decomposition. An application to undirected graphs was discussed in that paper as yet another special case. Subsequent research has focused on this undirected version, usually under the name split decomposition.

Let \( A \) and \( B \) be two sets of vertices containing the subsets \( A' \subseteq A \) and \( B' \subseteq B \). The composition operation defining split decomposition adds all edges \( \{a, b\} \) such that \( a \in A' \) and \( b \in B' \). An example appears in figure 2.2. In effect, we are forming the join of \( A' \) and \( B' \); hence the name join decomposition. But if we consider the inverse decomposition operation, the name split decomposition becomes more appropriate. A split in a graph is a partition \((A, B)\) of its vertices, \(|A|, |B| > 1\), for which there exist subsets \( A' \subseteq A \) and \( B' \subseteq B \), such that no edge exists between \( A \) and \( B \) except those between \( A' \) and \( B' \), and of these, all possible edges exist (see figure 2.2). Roughly speaking, split decomposition finds a split \((A, B)\) and recursively decomposes its parts \( A \) and \( B \).

Splits can be seen to generalize modules (cf. figures 2.1 and 2.2). Indeed, modular decomposition is another special case of Cunningham and Edmond’s decomposition theory. Furthermore, many of the applications of modular decomposition extend to split decomposition. The latter is employed in the recognition of distance-hereditary graphs [4, 87], parity graphs [32], and circle graphs [126]. We have already mentioned its use in solving restricted versions of combinatorial optimization problems [118]. Novel applications for split decomposition – like those observed for modular decomposition – include distance-labelling [74] and a relationship with hierarchical clustering exploited in [55].

There are also structural analogues between the two decomposition schemes. It is known, for instance, that circle graphs indecomposable with respect to split decomposition have unique rep-
Figure 2.2: Composing two graphs by adding all possible edges between \( A' = \{a, b, c, d\} \) and \( B' = \{e, f\} \). If \( A = \{a, b, c, d, k, \ell\} \) and \( B = \{e, f, g, h, i, j\} \), then \((A, B)\) is a split in the resulting graph.

representations [13, 43, 68]. The same is true of permutation graphs indecomposable with respect to modular decomposition (it is worth noting that circle graphs generalize permutation graphs). Similarly, the totally decomposable graphs with respect to split decomposition – the distance-hereditary graphs – generalize the totally decomposable graphs with respect to modular decomposition – the cographs. Moreover, cographs are precisely those graphs with clique-width 2, whereas distance-hereditary graphs are precisely those graphs with rank-width 1. Given the relationship between clique-width and rank-width [111], it is unsurprising that split decomposition plays a critical role in the recognition of graphs with clique-width \( \leq 3 \) [38]. Moreover, both modular decomposition and split decomposition can be seen as special cases of clique-width decomposition.

Algorithms computing split decomposition can also be categorized according to their approach. By computing split decomposition, we mean computing the split decomposition tree. Cunningham showed it can be reduced to a procedure that takes a pair of edges and determines if they cross a split [50]. Given such a procedure, it can be recursively called to decompose the parts of any split thus identified. The earliest polynomial time algorithms for split decomposition adopt this approach: first an \( O(n^4) \) algorithm by Cunningham [50], then \( O(n^3) \) by Bouchet [12], and finally an \( O(nm) \) algorithm by Gabor et al. [68]. Spinrad developed a variant in which the test is whether two (not necessarily adjacent) vertices are on the same side of a split [123]. This eventually led to an \( O(n^2) \) split decomposition algorithm he co-developed with Ma [98].

An entirely new perspective was provided by Dahlhaus, who claimed the first linear-time split decomposition in an extended abstract of 1994 [52]. The full journal paper didn’t appear until 2000 [55]. His algorithm uses modular decomposition as a subroutine, and more generally deviates from the approaches above by exploiting a relationship he observed between hierarchical clustering and split decomposition. That relationship manifests itself in the computation of overlap classes of
set systems. His two papers differ significantly, but it is difficult from reading either to appreciate the application of their ideas. The presentation in both is opaque, and understanding the algorithm in so far as verifying its correctness and running-time is extremely difficult.

Recently, two groups in France independently undertook to simplify the task of computing split decomposition in linear-time. One took the approach of closely following Dalhaus’ paper to better understand it. They first published an improved presentation and simplification of Dahlhaus’ subroutine for identifying overlap classes within set systems [28]. This led them to a deeper understanding of its role in computing split decomposition, and eventually to view Dahlhaus’ algorithm in terms of partitive families of sets, which were first defined in [29]. They recently produced a reimagining of Dahlhaus’ algorithm along these lines [27]. The new presentation is far simpler and easily verified.

The second group began under the assumption that the split decomposition tree itself was the source of the apparent difficulty. They claimed the algorithmic properties of split decomposition were still poorly understood, and that a potential roadblock lay in the split decomposition tree (pg. 42, [77]). A new combinatorial object called graph-labelled trees (GLTs) was proposed by them in response [77]. It amounts to a reorganization of the elements in Cunningham’s canonical split decomposition tree, and formalizes ideas originally presented in [64] for the drawing of distance-hereditary graphs.

An indication of the efficacy of GLTs is the fully-dynamic recognition algorithm for distance-hereditary graphs presented in [77]. Their is far simpler than the algorithm developed for the problem by the author in [129]. The latter uses a complicated, ad hoc data structure based on a Breadth-First Search layering of the graph. The author did not use the split decomposition tree because it conveys little information of the adjacencies in the underlying graph. GLTs provide a far more transparent encoding of split decomposition in this regard.

The split decomposition algorithm developed in this thesis (chapter 5) is an extension of the work started by the second French group. It is the first to compute split decomposition using GLTs. It should be noted that our algorithm and that of the first French group were developed simultaneously and independently. Ours takes a more algorithmic approach. We proceed incrementally, adding vertices according to an LBFS ordering. The incremental approach succeeds in being simpler than Dahlhaus’ algorithm, and the LBFS ordering guarantees its efficiency.

The incremental characterization of split decomposition we develop is defined in terms of the additional structure revealed by GLTs. This same structure helps us demonstrate new connections between LBFS and split decomposition. It also forms the basis of the charging argument we use to prove the algorithm’s running time. Our algorithm thereby demonstrates the advantages of GLTs
Figure 2.3: A chord diagram on the left, and its corresponding circle graph on the right.

and argues for their continued use in algorithmic applications of split decomposition. Indeed, as one early reviewer of the work commented, it can be argued that GLTs should be the way of viewing split decomposition.

2.3 Circle Graphs

Circle graphs are the intersection graphs of chords inscribing a circle (see figure 2.3). They were introduced by Even and Itai, who demonstrated a relationship between colouring circle graphs and sorting permutations using stacks [65]. An extension of this is an application to the shipping container stowage problem [3]. Their geometric interpretation makes them adaptable to many different settings. Another real-world application, for instance, is in constructing DNA sequences from shorter subsequences [1], while more theoretical applications include knot theory [30]. Some of their applications arise from an equivalent characterization in terms of double-occurrence words; in this context they are generally called alternance graphs. They were introduced under this name by Bouchet in proving a question of Gauss concerning self-intersecting closed curves in the plane [11]. Another name sometimes given them in this context is overlap graphs (e.g. [19]).

There has recently been renewed interest in circle graphs due to their connection to vertex-minors and rank-width. They are conjectured to play a role with respect to vertex minors and rank-width analogous to the role played by grids with respect to graph minors and tree-width [110]. This relationship arises from the definition of vertex minors in terms of local complementation, and the properties circle graphs demonstrate with respect to this operation. As reported by Bouchet [15], these connections were first observed by Kotzig in a series of seminars. They are at the core of many applications of circle graphs, including the Cycle Double Cover Conjecture [76], graph
polynomials [2], and gene assembly in ciliates [19]. Much of this rich theory can be traced to Bouchet’s seminal work on isotropic systems (e.g. [13, 14]).

One application of isotropic systems is a characterization of circle graphs in terms of forbidden vertex-minors [15]. This was recently extended to a characterization in terms of forbidden pivot-minors [75]. Although no forbidden induced subgraph characterization is known, recent progress in this direction appears in [57] and [9]. In particular, the former proves a conjecture that the Helly circle graphs are the diamond-free circle graphs. Bipartite circle graphs have been characterized by De Frayssieux in terms of planar matroids [60], which is used to prove a special case of the conjecture alluded to above. On the other hand, the recognition of circle graphs has mostly followed a characterization in terms of split decomposition [68, 13].

The polynomial-time recognition of circle graphs was first posed as an open-problem in the book by Golumbic [82]. It was also mentioned as such by Garey et al. in their paper proving that colouring circle graphs is NP-complete [72]. The problem received considerable attention in the years afterward, and was solved independently by Naji [108], Bouchet [13], and Gabor et al. [68], whose algorithms run in time $O(n^9)$, $O(n^4)$, and $O(nm)$, respectively. The $O(n^4)$ bound for Bouchet’s algorithm assumes it uses the procedure of Spinrad [123] to determine if a graph is indecomposable with respect to split decomposition. An $O(n^2)$ circle graph recognition algorithm was eventually developed by Spinrad himself in 1994 [126]. Whereas Naji’s algorithm is based on his characterization of circle graphs in terms of a set of linear equations (see e.g. [73]), the more efficient algorithms of Bouchet, Gabor et al., and Spinrad all depend on split decomposition.

It is known that a graph is a circle graph if and only if every part in its split decomposition is a circle graph [13, 68]. The three more efficient circle graph recognition algorithms each apply split decomposition as a preprocessing step. They thereby reduce the problem of recognizing circle graphs to one of recognizing circle graphs that are indecomposable with respect to split decomposition. Such graphs have the special property of having unique chord diagrams\(^1\), which was independently proved by Bouchet and Gabor et al. in their papers [13, 68].

In his algorithm, Bouchet applies local complementation to further reduce the problem of recognizing circle graphs. By contrast, Gabor et al. incrementally construct the unique chord diagram for the parts in the split decomposition that are indecomposable. Their idea is that the endpoints of chords partition the circle into a set of arcs. For each vertex not yet processed, they maintain the set of arcs where the endpoints of its corresponding chord may be placed. As each new chord is added, the set must be updated for those vertices still not processed. Spinrad borrows this idea

\(^1\)Up to some simple operations like reflection, rotation, scaling, etc.
for his algorithm, but instead maintains an “approximate” chord diagram at each step.

The circle graph recognition algorithm developed in this thesis (chapter 6) can be viewed in similar terms. However, we derive a new LBFS characterization of circle graphs that obviates the need to maintain the set of arcs just described. We show that if vertices are added according to an LBFS ordering, then the endpoints of each vertex’s neighbour will appear consecutively, in the sense of figure 2.4. This implicitly defines the appropriate set of arcs for each vertex. We define a new data-structure for chord diagrams that allows each chord to be placed in constant time.

Unfortunately, our use of LBFS complicates the reduction to indecomposable parts. To avoid having to compute an LBFS for each such part, we resort to GLTs and their encoding of split decomposition. Given an LBFS of the entire graph, the structure of GLTs can be used to induce an LBFS ordering for each part in the decomposition. The resulting algorithm is the first to improve Spinrad’s long-standing quadratic bound on circle graph recognition.

### 2.4 Lexicographic Breadth-First Search

*Breadth-First Search* (BFS) is traditionally implemented by a queue. As each new vertex is explored, its neighbours not already in the queue are added to the queue. The order in which these neighbours are added is left unspecified by the algorithm. By changing this ordering, we change the order in which vertices are explored in the BFS. Another way of viewing this is that these neighbours are all “tied” during their lifetime in the queue: their ordering within it is arbitrary, and changing it is still consistent with BFS.

*Lexicographic Breadth-First Search* (LBFS) is a refinement of BFS that breaks these ties according to a lexicographic labelling scheme. It allows the order of tied vertices to change during their lifetime in the queue. Roughly speaking, those with a greater propensity of neighbours earlier
in the queue will be made to appear earlier than their BFS tied cohorts, and will therefore be explored earlier.

Since its inception, LBFS has become a standard tool for working with restricted classes of graphs. It was designed by Rose, Tarjan, and Lueker in the 1970s to recognize chordal graphs in linear time [119]. They proved that LBFS applied to a graph produces a \textit{perfect elimination ordering} (PEO) if and only if the graph is chordal. Their chordal graph recognition algorithm simply applies LBFS and then determines if a PEO was produced, which can be done in linear-time.

This is typical of LBFS applications. It has frequently been used to recognize other classes of graphs in similar fashion: apply LBFS to produce a characteristic ordering, and then verify that ordering. The list of examples is lengthy and includes, amongst others: \textit{cographs} [18], \textit{distance-hereditary graphs} [17], \textit{interval graphs} [41], \textit{P}\textsubscript{4}-\textit{reducible graphs} [17], \textit{unit-interval graphs} [35], and \textit{bipartite permutation graphs} [26]. Compared to non-LBFS recognition algorithms for these classes, the LBFS versions are usually much simpler. This is one reason for the attempted generalization of this compute-then-check paradigm [58].

Despite its role in breaking ties, LBFS often fails to do so itself. This happens when vertices tied in the sense of BFS described above share the same neighbours amongst vertices earlier in the queue. Many refinements of LBFS have been proposed to break its ties. A simple example is where the vertex with the largest degree is chosen [16]. Some of the recognition algorithms cited above employ what has come to be known as a \textit{multi-sweep} approach to tie-breaking. Here, LBFS is applied several times in succession, and ties are resolved according to previous orderings. A detailed treatment of the subject can be found in the comprehensive survey of LBFS by Corneil [34].

Structural results based on LBFS tend to be stated in terms of LBFS \textit{end-vertices}, which are those vertices that appear last in an ordering produced by LBFS. Such vertices and their properties are explicitly studied in [39], and are investigated in the context of diameter approximation in [37, 61]. Their properties (with respect to interval graphs) are used by Korte and Mohring in [94] to streamline the incremental interval graph recognition of Booth and Lueker [10]. A kind of reverse application of this idea occurs in the distance-hereditary graph recognition algorithm of [95], which uses properties of LBFS end-vertices to guarantee the efficiency of its pruning strategy.

In general, results on LBFS end-vertices can be seen to encompass results characterizing LBFS orderings of different classes of graphs. Such results underpin the graph recognition paradigm described above, but also exist for classes of graphs whose recognition is independent of LBFS. \textit{Asteroidal triple-free} graphs are one example [40, 39], and \textit{house, hole, domino-free} graphs provide another [93]. The interested reader should refer to [16] for other instances.

The results in this thesis benefit from LBFS in ways that are consistent with the modes above.
For modular decomposition, we propose a new method of computing LBFS orderings based on recursion. We then modify the recursively computed orderings to produce a factorizing permutation. The modification that occurs afterward can be interpreted as a retroactive tie-breaking strategy. The efficiency of our incremental split decomposition algorithm depends on the properties of LBFS end-vertices, and is a kind of generalization of the result for distance-hereditary graph recognition already noted. Finally, we develop a new characterization of circle graphs in terms of LBFS end vertices and apply this characterization in their recognition. In each case, LBFS helps to simplify the algorithm involved.
Chapter 3

Technical Background

This chapter provides the definitions, notation, and background results that will be required by our treatment of the topics introduced in chapter 2: Modular Decomposition, Split Decomposition, Circle Graphs, and Lexicographic Breadth-First Search. Most of what appears here is repeated from the sources cited along with each topic in chapter 2, although in certain places we have altered the presentation to suit our needs.

In general, with few exceptions, definitions will not be sourced, nor simple properties that follow easily from these definitions, many of which might best be considered “folklore”. The simplest of these properties will be stated without proof as remarks, although sometimes an informal justification will accompany them. Lemmas stated without proof are due to other sources and will be duly cited. All other proofs are due to the author. We begin below with basic graph theory notation and terminology that will be used throughout the thesis. An index appears at the end of the thesis to assist the reader.

3.1 Preliminaries

Basic graph theory definitions, terminology, and notation can be found in [133], which should be consulted for anything not defined here. Unless otherwise specified, all graphs in this document are simple and undirected. The set of vertices of a graph $G$ is denoted $V(G)$, and the set of its edges is denoted $E(G)$. We use $n$ to represent $|V(G)|$ and $m$ to represent $|E(G)|$. The edge $\{x, y\}$ will be written as $xy$ for simplicity. An $xy$-path is a path where no vertices repeat and whose endpoints are $x$ and $y$. A connected component of a graph will be referred to as a component. A co-component of $G$ is a connected component of $\overline{G}$.

The neighbourhood of the vertex $x$ is denoted $N(x)$. The degree of a vertex $x$ is denoted $d(x)$. 
We let $N[x] = N(x) \cup \{x\}$ and call this the *closed-neighbourhood* of $x$. Two vertices $x$ and $y$ are *twins* if $N[x] = N[y]$. A vertex is *universal* in a graph if it is adjacent to every other vertex in the graph. A vertex is universal to a set of vertices if it is adjacent to every vertex in the set; a vertex is *isolated* from a set of vertices if it is adjacent to no vertex in the set.

For a subset $S \subseteq V(G)$, we use $G[S]$ to denote the subgraph induced by the vertices $S$. We use $G + (x, S)$ to denote the graph formed from $G$ by adding a new vertex $x$ whose neighbourhood is $S \subseteq V(G)$; when $S$ is implied or unimportant, we will simply say $G + x$. The graph $G - x$ is formed from $G$ by deleting the vertex $x$ along with all of its incident edges.

If $S \subset V(G)$ and $|S| > 1$, then we say that $S$ is *non-trivial*; otherwise $S$ is *trivial*. A bipartition $(A, B)$ of $V(G)$ is *non-trivial* if $|A|, |B| > 1$. Two sets $A$ and $B$ overlap if each of the following is non-empty: $A - B$, $A \cap B$, and $B - A$.

A *clique* is a graph where all possible edges are present. A *star* is a graph with at least three vertices, exactly one of which is universal, and having no edges other than those incident to its universal vertex. The universal vertex in a star is called its *centre*, while each of its other vertices is called an *extremity* of the star.

A *node* is a non-leaf vertex in a tree. *Internal tree edges* are those whose endpoints are both nodes. Given a tree $T$ and a subset of its leaves $S$, we use $T[S]$ to denote the smallest subtree of $T$ spanning the leaves $S$.

A *partition* of a set $S$ is a collection of disjoint, non-empty *partition classes* $\{P_1, \ldots, P_k\}$ such that $P_1 \cup \cdots \cup P_k = S$. An *ordered partition* of $S$ is a partition of $S$ in which the partition classes are ordered. Let $\mathcal{P}$ and $\mathcal{P}'$ be two partitions of a set $S$. Then $\mathcal{P}$ is a coarser partition than $\mathcal{P}'$ if for each partition class $P' \in \mathcal{P}'$ there is a partition class $P \in \mathcal{P}$ such that $P' \subseteq P$; conversely, $\mathcal{P}'$ is called a refinement of $\mathcal{P}$.

An *alphabet* is a set of symbols; these symbols are called letters. A *string* over the alphabet $\Sigma$ is a sequence of letters from $\Sigma$, each letter possibly appearing multiple times. We use $\epsilon$ to denote the string containing no letters. If $\pi$ is a string, then: $|\pi|$ is the number of letters in the string; $\pi^r$ is the string formed by reversing the order of the letters; and a *substring* of $\pi$ is any (possibly empty) sequence of consecutive letters in $\pi$. If $\pi = \alpha\beta$ is a string, where $\alpha$ and $\beta$ are substrings, then $\alpha$ is a *prefix* of $\pi$ and $\beta$ is a *suffix* of $\pi$; furthermore, $\pi' = \beta\alpha$ is a rotation of $\pi$. Finally, suppose that $\pi = \alpha\beta\gamma$ for substrings $\alpha$, $\beta$, and $\gamma$; if we are only interested in $\beta$, then we will simplify notation by writing $\pi = \cdots \beta \cdots$. 
3.2 Modular Decomposition

A module is a set $M \subseteq V(G)$ such that for each pair $x, y \in M$, we have $N(x) - M = N(y) - M$. The idea is that $x$ and $y$ are indistinguishable with respect to $V(G) - M$: either one can be used to “represent” $M$ with respect to $V(G) - M$. Of course, $V(G)$ and $\{x\}$ are always modules, for any $x \in V(G)$. A graph is *prime* if these are its only modules; in other words, a graph is prime if it only contains trivial modules.

A module is *strong* if no other module overlaps it. For example, if $G$ is disconnected, then each of its components is a strong module. Similarly, if $\overline{G}$ is disconnected, then each of $G$’s co-components is a strong module. Observe as well that, in both cases, any union of these (co-)components is a module of $G$. For this reason, $G$ is called *degenerate* in both cases. The union of two modules is not always a module, although one other exception is observed below:

**Remark 3.2.1.** Let $M$ and $M'$ be two overlapping modules of $G$. Then $M \cup M'$ is a module of $G$.

This gives rise to the notion of a module being *maximal*, which is the case when there is no non-trivial module properly containing it. In the examples above, the (co-)components of $G$ are its maximal strong modules.

Modular decomposition is the process of recursively decomposing a graph according to its maximal strong modules. To compute the modular decomposition of a graph is to compute its *modular decomposition tree* (MD tree), defined below:

**Definition 3.2.2.** Let $G$ be a graph. If $V(G) = \{x\}$, for some vertex $x$, then the MD tree for $G$ is simply the leaf $x$. If $|V(G)| > 1$, then let $M_1, \ldots, M_k$ be the maximal strong modules of $G$, and for each $M_i$, let $T_i$ be the MD tree for $G[M_i]$. Then the MD tree for $G$ is a rooted tree whose root node has the roots of the $T_i$’s as children.

Each node in the MD tree is labelled either as *parallel*, *series*, or *prime*, based on the following result:

**Lemma 3.2.3.** [71, 84] For any graph $G$, either $G$ is disconnected, $\overline{G}$ is disconnected, or there exists a set $U \subseteq V(G)$ and a unique partition $\mathcal{P}$ of $V(G)$ such that:

1. $|U| > 3$;
2. $G[U]$ is a maximal prime subgraph of $G$;
3. for each partition class $P \in \mathcal{P}$, $|P \cap U| = 1$. 

17
Figure 3.1: The modular decomposition tree for the graph in figure 2.1. We use “P” to stand for prime, and 0 and 1 to stand for “parallel” and “series”, respectively. The use of 0 and 1 is common in the literature.

It is not difficult to see that the unique partition in the third case is the set of maximal strong modules of $G$, and the set $U$ is formed by selecting a representative from each. The root of the MD tree for $G$ is labelled prime in this third case. When $G$ is disconnected, the root is labelled parallel, and when $G$ is disconnected, the root is labelled series. The other nodes are labelled recursively according to the definition of MD trees. A node’s label is called its type. See figure 3.1 for an example MD tree.

As a shorthand, we will often refer to $G$ as either parallel, series, or prime, depending on which of the cases in lemma 3.2.3 applies. Although this abus es our earlier definition of prime, our intent when using the term will be clear from context. Notice that a graph is degenerate if it is either parallel or series. Nodes will be referred to as either parallel, series, (degenerate,) or prime, depending on their type. We will often refer to a node as a shorthand for the set of its descendant leaves, with our desired meaning being clear from context. The next two results provide an example of these conventions:

**Lemma 3.2.4.** [105] Let $T$ be the modular decomposition tree for $G$, and let $M$ be a module in $G$. Then either:

1. there is a node in $T$ equal to $M$;

2. there is a degenerate node in $T$ and $k > 1$ of its children whose union equals $M$.
Remark 3.2.5. Let $M$ and $M'$ be two overlapping modules of $G$. Then there is a degenerate node $u$ in the MD tree for $G$ such that $M$ equals the union of $k > 1$ children of $u$, and $M'$ equals the union of $k' > 1$ children of $u$, and the two sets of children overlap.

The next remark is the converse of condition 2 of lemma 3.2.4. Although it was observed earlier, we collect it below for future reference:

Remark 3.2.6. Let $u$ be a degenerate node in the MD tree for $G$. Then any union of $u$’s children is a module in $G$.

Now, for $G$ to be disconnected, it must have at least two components; and for $\overline{G}$ to be disconnected, $G$ must have at least two co-components. Thus, the next result follows from condition 1 of lemma 3.2.3 and the correspondence between the set $U$ and the maximal strong modules of $G$:

Remark 3.2.7. Every node in the MD tree has at least two children.

Another way of viewing modular decomposition is in terms of permutations of $V(G)$:

Definition 3.2.8. A factorizing permutation of $G$ is any ordering of $V(G)$ in which the vertices in each strong module appear consecutively.

Given the MD tree for $G$, a factorizing permutation can be obtained by printing the leaves of the tree as they are encountered in a (pre/post/in)-order traversal of the tree. This thesis uses them for the converse: computing the tree from the permutation.

### 3.3 Split Decomposition

Splits generalize modules. Let $M$ be a module in the graph $G$. Then every vertex in $V(G) - M$ is either universal to $M$ or isolated from $M$. Splits make this relationship symmetric. Specifically, a split in a graph $G$ is a non-trivial bipartition $(A, B)$ of $V(G)$ such that there exists sets $A' \subseteq A$ and $B' \subseteq B$ where:

- every vertex $x \in A - A'$ is isolated from $B$;
- every vertex $x \in A'$ is universal to $B'$ and isolated from $B - B'$;
- every vertex $y \in B - B'$ is isolated from $A$;
- every vertex $y \in B'$ is universal to $A'$ and isolated from $A - A'$.
The sets $A'$ and $B'$ are called the frontiers of the split. The idea behind a split is that any vertex in $A'$ can be used to “represent” $A$ with respect to $B$. Similarly, any vertex in $B'$ can be used to “represent” $B$ with respect to $A$. See figure 2.2 for an example. One obvious remark is the following:

**Remark 3.3.1.** Let $(A, B)$ be a split in the graph $G$, and let $A'$ and $B'$ be the frontiers of the split. If $G$ is connected, then $A', B' \neq \emptyset$.

A graph without a split is called prime\(^1\). Cliques and stars are called degenerate\(^2\) because (if they have more than 3 vertices, then) any non-trivial bipartition of their vertices induces a split. The following split operation can also be defined:

**Definition 3.3.2.** Let $G$ be a graph having the split $(A, B)$ with frontiers $A'$ and $B'$. The split operation with respect to $(A, B)$ results in two graphs: $G_a = G[A] + (a, A')$ and $G_b = G[B] + (b, B')$, where $a$ and $b$ are two new vertices.

Split decomposition is defined by recursively applying the split operation until all graphs are prime. Once a graph has been decomposed, it can be retrieved by recursively applying the inverse of each split-operation, defined as follows:

**Definition 3.3.3.** The neighbourhood-join of $G_a$ and $G_b$ with respect to $a \in V(G_a)$ and $b \in V(G_b)$ is the graph $G$ where $V(G) = V(G_a) \cup V(G_b) - \{a, b\}$ and $E(G) = E(G_a) \cup E(G_b) \cup \{xy \mid x \in N(a), y \in N(b)\}$.

It happens that split decompositions are not unique. This arises because of induced cliques and stars, which as noted above, are highly decomposable. Uniqueness is obtained by defining a canonical decomposition. Consider instead the process of recursively decomposing (according to the split operation) until all graphs are either cliques, stars, or prime. Cunningham showed that every graph can be uniquely (minimally) decomposed in this way \([50]\). By a “minimal” decomposition, we mean one that can be obtained from another by application of the split operation. It is this latter decomposition into cliques, stars, and prime graphs that is thought of when referring to split decomposition. We will adopt this convention throughout the thesis. Thus, by its uniqueness, we can refer to the split decomposition of a graph.

Generally, to compute split decomposition means to compute the corresponding unique split decomposition tree. This tree is associated with the split decomposition of a graph as follows:

\(^1\)C.f. the definition of prime for modular decomposition. We will use “prime” in both its senses, the meaning being clear from context.
\(^2\)Again, we use “degenerate” here as well as for modular decomposition; the intended meaning will also be clear from context.
• the leaves of the tree correspond to the vertices of the graph, while the nodes correspond to
  the new vertices $a$ and $b$ created by each split operation;

• each pair $a$ and $b$ created by the same split operation are adjacent;

• leaves are adjacent to the new vertex $a$ (resp. $b$) created by the final split operation in which
  they participate;

• there are no other edges.

An example split decomposition tree appears in figure 3.2.

Notice that every internal tree edge in the split decomposition tree defines a bipartition of
$V(G)$ that induces a split in $G$. These are the splits according to which the graph was decomposed.
Unfortunately, it is unclear from the split decomposition tree what subsets of these leaves are the
frontiers of the split. That information must be retrieved by resorting to $G$. More generally, it is
unclear from the split decomposition tree which pairs of leaves are adjacent in $G$. An alternative
means of encoding split decomposition that does not suffer from this drawback is presented below.

### 3.3.1 Graph-Labelled Trees

Although we will use graph-labelled trees (GLTs) to encode split decomposition, they exist inde-
dependent of this encoding as the following combinatorial object:

**Definition 3.3.4.** [77] A graph-labelled tree (GLT) is a pair $(T, F)$, where $T$ is a tree and $F$ a set
of graphs, such that each node $u$ of $T$ has $d(u) > 2$ and is labelled by the graph $G(u) \in F$, and there
exists a bijection $\rho_u$ between the edges of $T$ incident to $u$ and the vertices of $G(u)$. See figure 3.3
for an example.
Figure 3.3: An example of a graph-labelled tree.

Discussion of GLTs is facilitated by the notation and terminology below, where \((T, \mathcal{F})\) is a GLT, \(u\) and \(v\) are adjacent nodes in \(T\), and \(e = uv\):

- the edge \(e\) is a tree-edge, whereas the edges in \(G(u)\) are label-edges;
- the vertices in \(G(u)\) are marker vertices.

- Let \(\rho_u(e) = q\) and \(\rho_v(e) = r\); then:
  - \(e\) is the edge corresponding to \(q\) (and \(r\));
  - \(q\) and \(r\) are the markers of \(e\);
  - \(q\) and \(r\) are opposite one another.

- If the leaf \(x\)'s incident edge is \(e\), and \(q \in V(u)\) is such that \(\rho_u(q) = e\), then \(x\) is opposite \(q\) (and vice versa).

- Notation will be simplified by letting \(V(u) = V(G(u))\);

- terminology will also sometimes be abused in using \(u\) as a shorthand for \(G(u)\), as in “...when \(u\) is prime...”; the intent will be clear from context.

The notion of induced subgraphs extends to GLTs as follows:
Definition 3.3.5. Let \((T, F)\) be a GLT, and let \(S\) be a subset of its nodes that induces a tree in \(T\). Then the GLT induced by \(S\) consists of the nodes in \(S\), their labels, and a leaf opposite every marker vertex corresponding to an edge \(uv \in E(T)\) such that \(u \in S\) and \(v \notin S\). See figure 3.4 for an example.

Two GLTs can be combined in the following way:

Definition 3.3.6. Let \(q\) be the marker vertex opposite the leaf \(x\) in the GLT \((T, F)\), and let \(q'\) be the marker vertex opposite the leaf \(x'\) in the GLT \((T', F')\). Grafting \((T, F)\) and \((T', F')\) with respect to \(q\) and \(q'\) creates a new GLT by deleting \(x\) and \(x'\) and adding a new edge having \(q\) and \(q'\) as its markers. See figure 3.5 for an example. For simplicity, we will sometimes refer to this as grafting \(q\) and \(q'\).

We alluded earlier to the fact that GLTs can be used to encode split decomposition. The means by which they do so is an accessibility relation defined by their labels.

Accessibility

The labels in a GLT exist to define the following accessibility relation between pairs of leaves and marker vertices:

Definition 3.3.7. \([77]\) Two marker vertices \(q\) and \(q'\) in a GLT are accessible from one another if there exists a sequence of marker vertices \(\Pi = q, \ldots, q'\) such that:
Two leaves \(x\) and \(y\) in a GLT are accessible from one another if their opposites are accessible from one another.

For example, in figure 3.3, \(h\) and \(e\) are accessible from one another, while \(h\) and \(a\) are not. Accessibility is fundamental for understanding GLTs. It is in this sense that GLTs encode graphs:

**Definition 3.3.8.** [77] Let \((T,F)\) be a GLT. Then its accessibility graph, denoted \(G(T,F)\) is the graph whose vertices are its leaves, with an edge between two distinct leaves \(\ell\) and \(\ell'\) if and only if they are accessible from one another. As an example, the accessibility graph of the GLT in figure 3.3 is the graph on the right in figure 2.2.

One obvious remark is the following:

**Remark 3.3.9.** The graph \(G(T,F)\) is connected if and only if every label in \(F\) is connected.

Notice that for each node \(u\) in the GLT \((T,F)\), the set of marker vertices \(V(u)\) can be seen to partition the leaves of \(T\). We formalize this idea in the definition below, and further refine the notion of accessibility:

---

Figure 3.5: Grafting two GLTs with respect to \(q\) and \(q'\).

- every pair of consecutive elements in \(\Pi\) are either the endpoints of a label edge or the markers of a tree edge;
- the edges thus defined alternate between tree edges and label edges.
Definition 3.3.10. Let \( q \in V(u) \) be a marker vertex in the GLT \( (T, F) \), and let \( e \) be the tree edge corresponding to \( q \). Then define \( L(q) \) to be the subset of leaves such that \( x \in L(q) \) if and only if the unique \( xu \)-path in \( T \) uses \( e \). Furthermore, \( A(q) \subseteq L(q) \) are those leaves in \( L(q) \) whose opposite are accessible to/from \( q \). We use \( T(q) \) to denote the subtree of \( T \) spanned by the leaves \( L(q) \).

For example, in the GLT of figure 3.3, if \( q \) is the centre of the star that is the label for the node incident to \( e \), then \( L(q) = \{a, b, c, d, g, h, i, j, k, \ell\} \) and \( A(q) = \{a, b, d, h, g, c\} \). The set \( A(q) \) is intended to represent the leaves that are accessible to/from \( q \). Based on this, we have the following:

Remark 3.3.11. If the graph \( G(T, F) \) is connected, then \( A(q) \neq \emptyset \) for every marker vertex \( q \) in \( (T, F) \).

Accessibility reveals how GLTs encode split decomposition:

Remark 3.3.12. Let \( e \) be an internal tree edge in a GLT whose accessibility graph is \( G \). Let \( q \) and \( r \) be \( e \)'s markers. Then \((L(q), L(r))\) is a split in \( G \) with frontiers \( A(q) \) and \( A(r) \).

The above remark is easily verified. It represents the analog of the comment we made earlier relating edges in a graph’s split decomposition tree to splits in that graph. One advantage of using graph-labelled trees instead of the split decomposition tree is that the adjacency relationship between leaves is directly encoded by the GLT through its accessibility relation: unlike the split decomposition tree, it is not necessary to resort to the graph to determine if two leaves are adjacent.

Unfortunately, graph-labelled trees do not uniquely encode graphs, as does the split decomposition tree. For example, the graph on the right in figure 2.2 could just as easily have been represented by a GLT with a single node labelled by the graph itself as opposed to the GLT in figure 3.3. To obtain uniqueness, we have to define a canonical form for GLTs.

The Split-Tree

The idea for obtaining uniqueness for GLTs is similar to the one used by Cunningham to obtain uniqueness of his split decomposition tree. The problem for Cunningham was the multiple ways in which cliques and stars could be split. For GLTs, this manifests itself through neighbouring cliques and stars. By removing certain types of neighbours, we can obtain unique GLT representations of graphs. We do so through the following operations:

Definition 3.3.13. Let \( e \) be a tree edge incident to nodes \( u \) and \( u' \) in a GLT, and let \( q \in V(u) \) and \( q' \in V(u') \) be the markers of \( e \). The node-join with respect to \( u \) and \( u' \) replaces \( u \) and \( u' \) with

\[\text{“A” for “accessible”}\]
a new node \( v \) labelled by the neighbourhood-join of \( G(u) \) and \( G(u') \) with respect to \( q \) and \( q' \). See figure 3.6.

**Definition 3.3.14.** The node-split operation is the inverse of the node-join. More precisely, let \( v \) be a node such that \( G(v) \) contains the split \((A, B)\). The node-split with respect to \((A, B)\) replaces \( v \) with two new adjacent nodes \( u \) and \( u' \) labelled by \( G_a \) and \( G_b \), the result of the split operation with respect to \((A, B)\), where the new vertices \( a \) and \( b \) are the markers of the tree edge thus created. See figure 3.6.

Of special interest are node-joins/splits involving degenerate nodes. The clique-join is a node-join involving adjacent cliques; its result is a node labelled by a clique. The clique-split is the inverse of the clique-join. The star-join involves adjacent stars whose shared tree-edge has exactly one marker that is the centre of a star; its result is a node labelled by a star. The star-split is its inverse operation. Figures 3.7 and 3.8 provide examples.

Notice that the accessibility graph remains unchanged following a node-join/split. By node-splitting any node that is not degenerate and clique/star-joining whenever possible, a unique GLT representation of the underlying accessibility graph is obtained:

**Definition 3.3.15.** [77] A GLT is reduced if all its labels are either prime or degenerate, and no clique-join or star-join is possible.
Theorem 3.3.16. [50, 78] For any connected graph $G$, there exists a unique, reduced graph-labelled tree $(T, \mathcal{F})$ such that $G = G(T, \mathcal{F})$.

The unique, reduced GLT guaranteed by the theorem above is called the split-tree and is denoted $ST(G)$. The split-tree encodes split decomposition (and all splits in a graph) in the following sense:

**Theorem 3.3.17.** [77] Let $(A, B)$ be a split in $G$. Then $ST(G)$ either:

1. contains an internal tree edge with markers $q$ and $r$ such that $A = L(q)$ and $B = L(r)$, or
2. permits a clique/star-split whose resulting internal tree edge has markers $q$ and $r$ such that $A = L(q)$ and $B = L(r)$.

Another way of looking at the previous theorem is as follows:

**Lemma 3.3.18.** Let $(A, B)$ be a split in $G$ with $x \in A'$ and $y \in B'$ on the frontiers of the split. Let $q_y$ be the marker vertex opposite $y$ in $ST(G[A \cup \{y\}])$, and let $q_x$ be the marker vertex opposite $x$ in $ST(G[B \cup \{x\}])$. Let $(T', \mathcal{F}')$ be the GLT that results from grafting $q_x$ and $q_y$, and let $e$ be the edge of $T'$ having $q_x$ and $q_y$ as its markers. Then either $(T', \mathcal{F}')$ is the split tree for $G$, or the split-tree for $G$ is obtained by a single star/clique-join involving $e$’s endpoints.

**Proof.** Let $G'$ be the underlying accessibility graph of $(T', \mathcal{F}')$. Observe that $G'$ is isomorphic to $G$. Also notice that every label in $\mathcal{F}'$ must be either prime or degenerate. Moreover, since $ST(G[A \cup \{y\}])$ and $ST(G[B \cup \{x\}])$ are both reduced, the only star/clique-join that is possible involves the endpoints of $e$.

A further advantage of a reduced GLT is its compactness:

**Lemma 3.3.19.** [77] Let $ST(G) = (T, \mathcal{F})$. If $x \in V(G)$, then $|T(N(x))| \leq 2 \cdot |N(x)|$.

Being reduced also means the following:
Remark 3.3.20. Let \((T, \mathcal{F})\) be a reduced GLT, and let \(q \in V(u)\) be one of its marker vertices. If \(d(q) = 1\), then \(u\) is a star.

Reviewing the definitions, it should be clear that the split tree merely reorganizes elements already in the split decomposition tree: the leaves in both are the vertices in the underlying graph; the marker vertices in the split tree correspond to the nodes in the split decomposition tree. The key difference between the two is that the labels in the split tree are induced by the adjacencies in the underlying graph. This direct representation of adjacencies is the primary advantage of the split-tree.

### 3.4 Circle Graphs

A **chord diagram** is a circle inscribed by a set of chords, none of which share an endpoint. A **circle graph** \(G\) is a graph for which there exists a chord diagram \(C\) such that:

- the chords of \(C\) can be identified with \(V(G)\);
- \(xy \in E(G)\) if and only if the chords corresponding to \(x\) and \(y\) intersect in \(C\).

We say that \(C\) is a **realizer** for \(G\); an example appears in figure 2.3. Circle graphs are also known as **alternance graphs** [11]. The latter name arises from an equivalent characterization in terms of **double occurrence words**:

**Definition 3.4.1.** A **double occurrence word** over the alphabet \(\Sigma\) is a string in which each letter in \(\Sigma\) appears exactly twice.

The graph \(G\) is an **alternance graph** if there exists a double occurrence word \(\pi\) over the alphabet \(V(G)\) such that \(xy \in E(G)\) if and only if there is a rotation of \(\pi\), call it \(\tau\), such that \(\tau = \cdots x \cdots y \cdots x \cdots y \cdots \). In other words, edges in alternance graphs are defined by alternating instances of letters in a double occurrence word; hence the name. The string \(\pi\) is also known as a **realizer** for \(G\).

When referring to double occurrence words, it will always be in the context of realizing some alternance graph \(G\), and the alphabet \(V(G)\) will be implied. As we have done so far, lowercase Greek characters will be used to denote double occurrence words and their substrings, while Roman characters will be used to denote letters (i.e. vertices) in double occurrence words. If \(\pi = \alpha \beta\) is a realizer, and \(x\) is a letter in \(\alpha\) (i.e. \(x\) is a vertex in \(G\)), then we will abuse notation somewhat and say \(x \in \alpha\).
The equivalence between circle graphs and alternance graphs arises from the obvious correspondence between endpoints of chords and letters in the double occurrence word. In this thesis we will use both characterizations interchangeably, depending on the setting. Chord diagrams will primarily be used for illustrative purposes like figures, and double occurrence words for more technical ends like proofs. When we say “realizer”, the meaning will be clear from context.

The following remark easily follows by removing the necessary chords/letters from a realizer:

**Remark 3.4.2.** If $G$ is a circle graph and $S \subseteq V(G)$, then $G[S]$ is a circle graph.

If $\pi$ is a realizer and $S \subseteq V(G)$, then $\pi[S]$ denotes the realizer formed from $\pi$ by removing all occurrences of vertices not in $S$. Clearly, $\pi[S]$ is a realizer for the circle graph $G[S]$.

Realizers are not generally unique. For example, if $\pi$ is a realizer for $G$, then so is $\pi^r$, and so is any rotation of these two realizers. These correspond to reflections and rotations of chord diagrams realizing $G$. This thesis will take these equivalences for granted:

**Convention 3.4.3.** If $\pi$ is a realizer for $G$, then any property that can be obtained by a rotation or reversal of $\pi$ will be assumed of $\pi$ itself.

As an example of the above convention, consider a realizer $\pi$ of $G$ and some $x \in V(G)$. Then we can assume, if necessary, that $\pi = x \cdots$, since otherwise a suitable rotation of $\pi$ would produce a realizer in which $x$ appears first. Another example concerns two non-adjacent vertices $x$ and $y$. Then either $\pi = \cdots x \cdots x \cdots y \cdots y \cdots$ or $\pi = \cdots x \cdots y \cdots y \cdots x \cdots$. We can assume either case for $\pi$ since a suitable rotation would suffice to get one from the other. Adopting convention 3.4.3 significantly simplifies the proofs concerning circle graphs in this thesis.

When we use the expression “degenerate” in the context of circle graphs, we mean degenerate in the sense of split decomposition; similarly for “prime”. Regarding the former, we observe the following, evidenced by figure 3.9:

**Remark 3.4.4.** If $G$ is degenerate, then $G$ is a circle graph.

Prime circle graphs are interesting in that they have unique realizers (up to convention 3.4.3):

**Lemma 3.4.5.** [13, 68, 44] If $\pi$ and $\pi'$ are realizers for the prime graph $G$, then there exists a rotation of $\pi$, call it $\tau$, such that either $\pi' = \tau$ or $\pi' = \tau^r$.

It will sometimes be helpful to differentiate the two occurrences of each vertex in a double occurrence word. When necessary, we will assign subscripts to letters for this purpose, as in $x_1$ and $x_2$. When we say $x_i$, it is with the understanding that $i \in \{1, 2\}$; and when $x_i$ and $x_j$ are
Figure 3.9: Chord diagrams for two degenerate graphs: on the left, a clique on four vertices; on the right, a star with three extremities. Chord diagrams can be obtained for any degenerate graph by following the patterns above.

referenced together, it is with the understanding that \{i, j\} = \{1, 2\}. If \(\pi = \alpha\beta\), and we say \(x_1 \in \beta\), it is only to say that (at least) one instance of \(x\) appears in \(\beta\); in particular, it does not mean that the first instance of \(x\) appears in \(\beta\). Assigned indices carry no meaning other than to distinguish between the two letters corresponding to each vertex. This will be crucial for understanding the proofs concerning circle graphs that follow in this thesis.

3.5 Lexicographic Breadth-First Search

Lexicographic Breadth-First Search (LBFS) is stated here as algorithm 3.5.1. An LBFS of \(G\) is an ordering of \(V(G)\) produced by algorithm 3.5.1. Let \(\sigma\) be an LBFS of \(G\), and consider some subset \(S \subseteq V(G)\). Then \(\sigma[S]\) is an ordering of \(S\) obtained from \(\sigma\) by removing all vertices not in \(S\). If \(a\) appears before \(b\) in \(\sigma\), then we write \(a <_\sigma b\). A vertex \(x \in V(G)\) is an \textit{end-vertex} for \(G\) if there exists an LBFS of \(G\) in which \(x\) appears last.

\begin{algorithm}
\caption{\textit{LBFS}(\(G\))}
\textbf{Input}: A graph \(G\) with \(n\) vertices.
\textbf{Output}: A numbering \(\sigma\) of \(V(G)\) inducing the ordering: \(\sigma^{-1}(1), \ldots, \sigma^{-1}(n)\).

\begin{algorithmic}
\ForAll{} \(x \in V(G)\) \textbf{do} \label{} \text{label}(x) \leftarrow \text{null};
\For{} \(i = 1\) \textbf{to} \(n\) \textbf{do}
\State pick an unnumbered vertex \(x\) with lexicographically largest label;
\State \(\sigma(x) \leftarrow i\) ; // \textit{assign} \(x\) \textit{the number} \(i\)
\ForAll{} unnumbered vertex \(y \in N(x)\) \textbf{do} \text{append} \(n - i + 1\) to \text{label}(y);
\EndFor
\EndFor
\end{algorithmic}
\end{algorithm}
As explained in chapter 2, LBFS orderings are just special cases of Breadth-First Search orderings. Hence:

**Remark 3.5.1.** Let $\sigma = x_1, \ldots, x_n$ be an LBFS of the graph $G$, and let $G_i = G[\{x_1, \ldots, x_i\}]$. If $G$ is connected, then so is $G_i$, for all $i$. Moreover, $\sigma[V(G_i)]$ is an LBFS of $G_i$.

One of the key notions for LBFS introduced in chapter 2 was the idea of vertices being tied. This is captured in the LBFS context by the notion of slices:

**Definition 3.5.2.** Consider an execution of algorithm 3.5.1 producing the ordering $\sigma$. The set $S(y)$ is the set of vertices having the lexicographically largest label at the time $y$ is assigned a number; note that $y \in S(y)$, by convention. We call each such set a slice in $\sigma$.

The next result is obvious:

**Remark 3.5.3.** Let $S$ be a slice generated by an execution of algorithm 3.5.1 that results in the ordering $\sigma$. Then the vertices in $S$ appear consecutively in $\sigma$.

The following lemma characterizes LBFS orderings:

**Lemma 3.5.4.** [62] An ordering $\sigma$ of a connected graph $G$ is an LBFS ordering if and only if for any triple of vertices $a <_\sigma b <_\sigma c$ with $ac \in E(G)$, $ab \notin E(G)$, there is a vertex $d <_\sigma a$ such that $db \in E(G)$, $dc \notin E(G)$.

The next two results follow from the one above:

**Lemma 3.5.5.** [85] Let $\sigma$ be an LBFS of a connected graph $G$, and let $M$ be a module of $G$, and suppose that $x \in M$. Then $\sigma[M]$ is an LBFS of $G[M]$.

**Lemma 3.5.6.** Let $\sigma$ be an LBFS of a graph $G$, and assume that $x \in V(G)$ appears last in $\sigma$. Assume there is a $k$ such that the set of the last $k$ vertices in $\sigma$ is a module $M$ in $G$. Then $\sigma[(V(G) - M) \cup \{x\}]$ is an LBFS of $G'(V(G) - M) \cup \{x\}]$.

Proof. If $|M| = 1$ then the result follows by assumption; so assume otherwise. For simplicity, let $G' = G'[V(G) - M] \cup \{x\}$, and let $\sigma' = \sigma[V(G')]$. Assume for contradiction that $\sigma'$ is not an LBFS of $G'$. Then by lemma 3.5.4, there are vertices $a <_{\sigma'} b <_{\sigma'} c$ with $ac \in E(G')$, $ab \notin E(G')$, and there is no vertex $d <_{\sigma'} a$ such that $db \in E(G')$ but $dc \notin E(G')$.

Of course, $a <_\sigma b <_\sigma c$, and $ac \in E(G)$ and $ab \notin E(G)$. So by lemma 3.5.4 again, there is a $d <_\sigma a$ such that $db \in E(G)$ but $dc \notin E(G)$. Obviously, $d \in M$ and $d \neq x$. Moreover, $x$ must be one of $a, b$, and $c$. But this contradicts our assumption that $M$ is the set of the last $k$ vertices in $\sigma$. □
One common implementation of algorithm 3.5.1 uses the partition refinement paradigm [85]. In this implementation, an ordered partition is maintained for all unnumbered vertices. The partition classes correspond to sets of vertices with the same label. A partition class $P$ appears before another class $P'$ in the ordering if and only if the vertices in $P$ have larger labels than those in $P'$. Thus, the first class defines the slice at each iteration of the algorithm. An arbitrary vertex $y$ is removed from that first class and numbered; then the remaining classes are refined according to $N(y)$: each class $P$ is replaced by $N(y) \cap P, P - N(y)$ in order. In this way, the invariant governing the order of classes is maintained. We will use a variant of this implementation in the modular decomposition algorithm presented in the next chapter.
Chapter 4

Simple, Linear-Time Modular Decomposition

Algorithmic applications of modular decomposition abound. This chapter responds to this need by developing a modular decomposition algorithm that is optimal, practical, and simple: it runs in linear-time, can be implemented with trees, and is no more conceptually challenging than the basic definition of a module. Theoretically, the algorithm is interesting for unifying two separate approaches previously employed for the problem: divide-and-conquer and factorizing permutations.

Previous divide-and-conquer approaches are reformulated here using LBFS: the problem is broken down according to the slices of an LBFS starting from some vertex \( x \). MD trees are recursively computed for the graph induced by each slice. We then apply partition refinement techniques to these trees, refining them to identify the strong modules not containing \( x \). A factorizing permutation results from the refinement, and this allows us to identify the strong modules containing \( x \). The two are then assembled to form the MD tree.

It is important to note here that the graphs in this chapter may be disconnected. This reflects the recursive nature of the algorithm. Although the input graph may be connected, we cannot guarantee that all of the induced subgraphs encountered during the recursion will also be connected.

The graph in figure 4.1 will be used throughout the chapter to demonstrate the algorithm’s operation. This chapter is based on [130], jointly developed with Derek Corneil, Michel Habib, and Christophe Paul.

4.1 Dividing by LBFS

The subproblems solved by our MD algorithm are defined by maximal LBFS slices:
Figure 4.1: An example graph to demonstrate the MD algorithm’s operation.

**Definition 4.1.1.** Consider the set of slices arising from some execution of algorithm 3.5.1 on the graph $G$, and let $S$ be one such slice. We say that $S$ is a maximal slice if $S \neq V(G)$, and there is no other slice $S'$ such that $S \cap S' \neq \emptyset$.

By remark 3.5.3, any LBFS defines an ordering on its maximal slices. This leads to the following definition:

**Definition 4.1.2.** Let $\sigma$ be an LBFS ordering of $G$ in which the vertex $x \in V(G)$ appears first. Let $S_1, \ldots, S_k$ be the ordering of the maximal slices defined by $\sigma$. Then $\{x\}, S_1, \ldots, S_k$ is the ordered maximal slice partition of $G$ with respect to $\sigma$. For simplicity, these will usually be referred to as maximal slice partitions, leaving both the ordering of the classes and $\sigma$ implied.

As an example, notice that $x, a, b, c, d, e, z, f, g, h, i, j, k$ is an LBFS ordering of the graph in figure 4.1. The corresponding maximal slice partition is: $\{x\}, \{a, b, c, d\}, \{e, y, z, f, g\}, \{h, i, j, k\}$. The following properties of maximal slice partitions can be verified for this example:

**Remark 4.1.3.** Let $P_1, \ldots, P_k$ be an ordered maximal slice partition of $V(G)$. Then:

1. $|P_1| = 1$;

2. for every pair $P_i, P_j, i < j$, each vertex $y \in P_i$ is either universal to or isolated from $P_j$;

3. for every pair $P_i, P_j, 1 < i < j$, there exists a $P_\ell, 1 \leq \ell < i$, and a $y \in P_\ell$ such that:
   
   (a) $y$ is universal to $P_i$ and isolated from $P_j$, and
   
   (b) every vertex in an earlier partition class than $P_\ell$ is either universal to both $P_i$ and $P_j$ or isolated from both $P_i$ and $P_j$.

The previous remark will form the basis of the proofs that follow in this chapter. We will use the next definition to parameterize the running-time of our algorithm:
Definition 4.1.4. Let $P_1, \ldots, P_k$ be a maximal slice partition of $G$. Then:

1. the singleton set $P_1$ and its lone vertex will be called the pivot; which one – the set or the vertex – we are referring to will be clear from context;

2. an edge of $G$ is active if its endpoints reside in different $P_i$’s;

3. each vertex $y \in P_i$ has an active list, denoted $\alpha(y)$, consisting of all $z \in N(y)$ such that $z \in P_j, j < i$.

The notion of pivot is borrowed from partition refinement. This is not by accident. We will use partition refinement to compute maximal slice partitions. The idea is to modify the traditional partition refinement implementation of algorithm 3.5.1 (see chapter 3). In our case, we start with a single partition class corresponding to $V(G)$. Then an arbitrary vertex is selected, call it $x$. We then compute the ordered partition: $x, N(x), V(G) - N[x]$. Notice that $N(x)$ is a slice in any LBFS starting from $x$. Moreover, it is the first maximal slice in a maximal slice partition having $x$ as pivot.

Our algorithm now recursively computes the MD tree for $N(x)$. As it does so, it will be recursively computing maximal slice partitions of subgraphs of $G[N(x)]$. For each of these, there will be a pivot. Just as we did above for $x$, this pivot will begin by splitting its partition class according to its neighbourhood. The key for computing a maximal slice partition will be to split other partition classes in the same way. Thus, after the recursion has finished and the MD tree for $G[N(x)]$ has been computed, we will be left with an ordered partition of the form: $x, N(x), P_1, \ldots, P_k$, where:

1. each $y \in N(x)$ is either universal to or isolated from $P_i$, for all $i$;

2. for every $P_i, i < k$, there is a $y \in N(x)$ such that $y$ is universal to $P_i$ and isolated from $P_{i+1}$.

Notice now that $P_1$ is a slice in some LBFS starting from $x$. Moreover, it is the second maximal slice in a maximal slice partition having $x$ as pivot. Our algorithm now recursively computes the MD tree for $G[P_1]$, and refines $P_2, \ldots, P_k$ in similar fashion. And so on. Along the way, it will have identified every maximal slice in some maximal slice partition having $x$ as pivot. The LBFS underlying the maximal slice partition in question is defined by the order in which vertices are selected as pivots. More important for our purposes, however, is that the MD trees for each maximal slice will have been computed. The rest of this chapter is dedicated to showing how these can be combined to form the MD tree for $G$.

The process just described is formalized as algorithm 4.1.1. The sets $P$ chosen during the while loop correspond to the maximal slices for the graph $G[S]$. To compute the MD tree for $G$, simply...
call algorithm 4.1.1 with the input \((G, \emptyset)\). The process by which the MD trees for each maximal slice are combined to form the MD tree for \(G\) is summarized as algorithm 4.5.1. We use it as a placeholder for now.

**Lemma 4.1.5.** Algorithm 4.1.1 is correct (up to the correctness of algorithm 4.5.1).

*Proof.* The proof is by induction on the size of \(S\). Assume algorithm 4.1.1 is correct when input any set with fewer elements than \(S\). Let \(P_1, \ldots, P_k\) be the sets removed from \(P\) during the while loop. By definition, each \(P_i \subseteq S\). The proof now easily follows by iteratively applying the induction hypothesis for each iteration of the while loop.

The rest of the chapter builds toward the specification of algorithm 4.5.1. Its input is defined below:

**Definition 4.1.6.** An ordered list of rooted trees \(T\) whose leaf sets define an ordered partition of \(V(G)\), for some graph \(G\), is called an ordered tree partition of \(G\). If that ordered partition of \(V(G)\) is a maximal slice partition of \(G\), then \(T\) is called a maximal slice tree partition of \(G\). In this case, the first tree consists only of a single leaf, which we call the pivot.

One of the key observations informally observed above is the following, which will appear in the proofs supporting the development of algorithm 4.5.1:

**Remark 4.1.7.** Let \(P_1, \ldots, P_k\) be a maximal slice partition of \(G\), where \(P_1 = \{x\}\) is the pivot. If \(N(x) \neq \emptyset\), then \(P_2 = N(x)\).

Finally, notice that we have algorithm 4.1.1 compute the active lists for its vertices. We do so because they can be used to refine the recursively computed MD trees in order to identify the strong modules not containing the pivot. From these, we can then identify the strong modules containing the pivot. The two types of modules can then be assembled to construct the MD tree for the entire graph. All of this is the work of algorithm 4.5.1. We start its development in the next section, looking at how modules not containing the pivot are identified.

### 4.2 Modules Not Containing the Pivot

Let \(T_1, \ldots, T_k\) be a maximal slice tree partition of \(G\), with \(T_1 = x\) the pivot. Let \(P_1, \ldots, P_k\) be the underlying maximal slice partition, and assume that each \(T_i\) is the MD tree for \(G[P_i]\). To find the modules of \(G\) not containing \(x\), we apply the following result:
Algorithm 4.1.1: \textit{DivideMDTree}(S, \mathcal{P})

\textbf{Input}: A non-empty set \( S \subseteq V(G) \), for some graph \( G \), and an ordered partition \( \mathcal{P} \) of a non-empty set \( S' \subseteq V(G) \), where \( S \cap S' = \emptyset \).

\textbf{Output}: A pair \((T, \mathcal{P}')\), where \( T \) is the MD tree for \( G[S] \), and \( \mathcal{P}' = P_1', \ldots, P_k' \) is a refinement of \( \mathcal{P} \) such that:

1. each vertex \( x \) will have had its active list \( \alpha(x) \) computed;

2. every vertex in \( S \) is either universal to or isolated from every \( P_i' \);

3. for every pair \( P_i', P_j', i < j \), such that \( P_i' \cup P_j' \subseteq P \) for some \( P \in \mathcal{P} \), there is a vertex in \( S \) that is universal to \( P_i' \) and isolated from \( P_j' \).

choose some \( x \in S \);

add \( x \) to \( \alpha(y) \), for each \( y \in N(x) \cap (S \cup S') \);

\textbf{foreach} class \( P \in \mathcal{P} \) \textbf{do}

\hspace{1em} \( A \leftarrow P \cap N(x) \); \( B \leftarrow P - A \);

\hspace{1em} \textbf{if} \( A, B \neq \emptyset \) \textbf{then} replace \( P \) in \( \mathcal{P} \) with \( A, B \) in this order;

\textbf{endfch}

\textbf{if} \( S = \{x\} \) \textbf{then} \textbf{return} \((x, \mathcal{P})\);

\textbf{if} \( S - (N[x] \cap S) \neq \emptyset \) \textbf{then} prepend \( S - (N[x] \cap S) \) to \( \mathcal{P} \);

\textbf{if} \( N(x) \cap S \neq \emptyset \) \textbf{then} prepend \( N(x) \cap S \) to \( \mathcal{P} \);

initialize the ordered list of trees \( T \) with \( \{x\} \);

\textbf{while} \( P \subseteq S \), \textit{where \( P \) is the first class in \( \mathcal{P} \)} \textbf{do}

\hspace{1em} remove \( P \) from \( \mathcal{P} \);

\hspace{1em} \((T, \mathcal{P}) \leftarrow \text{DivideMDTree}(P, \mathcal{P})\);

\hspace{1em} append \( T \) to \( T \);

\textbf{endw}

\( T \leftarrow \text{ConquerMDTree}(T); // \text{Algorithm 4.5.1} \)

\textbf{return} \((T, \mathcal{P})\);
Lemma 4.2.1. Let $P_1, \ldots, P_k$ be a maximal slice partition of $G$, with $P_1 = \{x\}$ the pivot. Let $M$ be a module not containing $x$. Then there is a $P_i, i > 2$, such that $M \subseteq P_i$, and $M$ is a module of $G[P_i]$.

Proof. It is clear that if $M \subseteq P_i$, then $M$ is a module of $G[P_i]$. So suppose for contradiction that there are two sets $P_i, P_j, i < j$, such that $M \cap P_i \neq \emptyset$ and $M \cap P_j \neq \emptyset$. As $x \notin M$ we can assume that $i > 1$. The contradiction now follows from condition 3(a) of remark 4.1.3.

Conversely, if $M \subseteq P_i$ is a module of $G[P_i]$, and $i > 2$, then $x \notin M$. Thus, identifying the modules of $G$ not containing $x$ reduces to identifying those modules of the $G[P_i]$’s that are not also modules of $G$. Of course, if $M \subseteq P_i$, and $y \in P_{\ell}, \ell < i$, then we already know that $y$ is either universal to or isolated from $M$, by condition 2 of remark 4.1.3. So to determine if $M$ is not also a module of $G$, it suffices to consider the active lists of vertices in $P_j, j > i$. This is the approach taken below by algorithm 4.2.1 to identify the modules of $G$ not containing $x$. Its output is a refinement of the $T_i$’s:

Definition 4.2.2. Given two rooted trees $T$ and $T'$, both containing the same set of leaves, we say that $T'$ is a refinement of $T$ if for every node $u$ in $T$, there is a node $v$ in $T'$ such that the leaves descending from $u$ and $v$ are the same. Given two ordered lists of rooted trees $T = T_1, \ldots, T_k$ and $T' = T'_1, \ldots, T'_k$, we say that $T'$ is a refinement of $T$ if each $T'_i$ is a refinement of $T_i$.

Suppose that $u$ is a degenerate node in $T_i$. Then by remark 3.2.6, any union of its children is a module in $G[P_i]$. However, not all of these will be modules in $G$. Algorithm 4.2.1 inserts new nodes into $T_i$ to isolate those that are from those that are not. It additionally gives labels of left or right to these new nodes. The labels will be used later to help identify the modules of $G$ containing $x$. An example of algorithm 4.2.1 is provided by figures 4.2 and 4.3.

Lemma 4.2.3. Algorithm 4.2.1 is correct.

Proof. It is clear that $T'$ is a refinement of $T$. It is also clear that any node labelled dead has exactly two children, one its left child, the other its right child.

So consider a node $u$ in $T'_i$ such that neither $u$ nor any of its descendants are labelled as dead. Assume for contradiction that there is a leaf $x \in L_j, j > i$, such that $x$ is neither universal to $u$ nor isolated from $u$. Then when $x$ comes under consideration in the second for loop, some leaf descendent from $u$ will be marked. As $x$ is not universal $u$, it follows that $u$ itself will not be marked. Thus, either $u$ or one of its descendants will be unmarked but have a marked child. Thus, $u$ or one of its descendants will be labelled as dead, a contradiction.
Algorithm 4.2.1: TreeRefinement($T$)

**Input:** A maximal slice tree partition $T = T_1, \ldots, T_k$ of some graph $G$, such that if $L_1, \ldots, L_k$ are the corresponding leaf sets, then $T_i$ is the MD tree for $G[L_i]$. Moreover, we assume that each leaf $x \in L_i$ has an associated set $\alpha(x)$ consisting of its neighbours amongst the leaves of the $T_j$'s, $j < i$. Finally, no node is labelled *dead* (see output).

**Output:** A refinement $T' = T'_1, \ldots, T'_k$ of $T$, in which some nodes are labelled *dead*, such that:

1. each node labelled *dead* has exactly two children, one the *left child*, the other the *right child*;

2. for each node $u$ in $T'_i$, every leaf $x \in L_j, j > i$, is either universal to or isolated from $u$ if and only if neither $u$ nor any of its descendants are labelled *dead*.

```
foreach $i = 2, \ldots, k$ do
  foreach $y \in L_i$ do
    foreach $j = 1, \ldots, i - 1$ do
      mark each leaf in $\alpha(y) \cap L_j$;
      while there is an unmarked node $u$, all of whose children are marked do mark $u$;
      foreach unmarked node $u$ with a marked child do
        let $A$ be the set of marked children of $u$, and let $B$ be its other children;
        if $|A| > 1$ and $u$ is degenerate then replace the children in $A$ with a new marked node inheriting $u$’s type, whose children are those nodes in $A$;
        if $|B| > 1$ and $u$ is degenerate then replace the children in $B$ with a new unmarked node inheriting $u$’s type, whose children are those nodes in $B$;
        if $u$ is not labelled dead then
          label $u$ as dead;
          if $j = 2$ then
            make $u$’s marked child its *left child*;
            make $u$’s unmarked child its *right child*;
          else
            make $u$’s marked child its *right child*;
            make $u$’s unmarked child its *left child*;
        endif
      endif
  endforeach
  Clear all marks off nodes (but not *dead* labels);
endfor
return $T'$;
```
Figure 4.2: Recursively computed MD trees for each maximal slice in the maximal slice partition \( \{x\}, \{a, b, c, d\}, \{e, y, z, f, g\}, \{h, i, j, k\} \) of the graph in figure 4.1.

Figure 4.3: The recursively computed MD trees of figure 4.2, after being refined according to algorithm 4.2.1. The darkly shaded nodes are those labelled \textit{dead}. Nodes labelled \textit{right} are those with diagonal shading up to the right, while those nodes labelled \textit{left} are diagonally shaded up to the left.
Now assume that every leaf \( x \in L_j, j > i, \) is either universal to or isolated from \( u. \) Then obviously whenever any descendant of \( u \) is marked, so too will be \( u. \) Hence, \( u \) will never be labelled as \textit{dead}.

Together, the next two results prove that algorithm 4.2.1 has correctly identified the modules of the \( G[P_i] \)'s that do not remain modules in \( G:\)

**Lemma 4.2.4.** Let \( T' = T'_1, \ldots, T'_k \) be the maximal slice tree partition of \( G \) output by algorithm 4.2.1. If \( u \) is a node in \( T'_i \) that is labelled dead, then neither \( u \) nor any of its ancestors is a module of \( G. \)

\textit{Proof.} The node \( u \) is labelled as \textit{dead} when there is a leaf \( y \) in a tree \( T'_j, j > i, \) that is adjacent to some, but not all, of the leaves descendent from \( u. \) Hence, \( u \) cannot be a module of \( G. \) Of course, \( y \) is also adjacent to some, but not all, of the leaves descendent from \( v, \) an ancestor of \( T'_i. \) Hence, \( v \) cannot be a module of \( G. \)

**Lemma 4.2.5.** Let \( T = T_1, \ldots, T_k \) be the maximal slice tree partition input to algorithm 4.2.1, and let \( T_1 = x \) be the pivot. Let \( T' = T'_1, \ldots, T'_k \) be the refinement of \( T \) output by algorithm 4.2.1. Let \( u \) be a node in \( T'_i \) such that neither \( u \) nor any of its descendants are labelled dead. Then \( u \) corresponds to a module of \( G \) not containing \( x, \) call it \( M, \) and the subtree rooted at \( u \) corresponds to the MD tree for \( G[M]. \)

\textit{Proof.} Let \( L_1, \ldots, L_k \) be the leaf sets of \( T; \) recall that these form a maximal slice partition. By lemma 3.2.4, either \( u \) was a node in \( T_i, \) or it corresponds to the union of \( k > 1 \) children of a degenerate node in \( T_i. \) Either way, \( u \) corresponds to a module of \( G[L_i], \) in the latter case by remark 3.2.6. So by condition 2 of remark 4.1.3, and output condition 2 of algorithm 4.2.1 (and lemma 4.2.3), we conclude that \( u \) is also a module of \( G. \) Let \( M \) be the module of \( G \) corresponding to \( u. \) By definition of MD trees, if the subtree rooted at \( u \) did not correspond to the MD tree for \( G[M], \) then \( T_i \) could not be the MD tree for \( G[L_i]. \)

Now, if \( M \) is a module of \( G \) not containing \( x, \) then it is either strong or it is not. In the former case, it corresponds to a node in the MD tree for \( G. \) In the latter case, by lemma 3.2.4, it corresponds to the union of \( k > 1 \) children of a degenerate node in the MD tree for \( G. \) Of course, by definition, that degenerate node must be strong. Thus, identifying the modules of \( G \) not containing \( x \) reduces to identifying the \textit{strong} modules of \( G \) not containing \( x. \) Therefore, the next result proves that the approach adopted by algorithm 4.2.1, of identifying the modules of the \( G[L_i] \)'s that do not remain modules in \( G, \) correctly identifies the module of \( G \) not containing \( x: \)
Lemma 4.2.6. Let $T = T_1, \ldots, T_k$ be the maximal slice tree partition input to algorithm 4.2.1, and let $T_1 = x$ be the pivot. Let $T' = T'_1, \ldots, T'_k$ be the refinement of $T$ output by algorithm 4.2.1. If $M$ is a strong module of $G$ not containing $x$, then there is a tree $T'_i$, and a node $u$ in $T'_i$, such that neither $u$ nor any of its descendants are labelled dead, and the subtree rooted at $u$ corresponds to the MD tree for $G[M]$.

Proof. Let $L_1, \ldots, L_k$ be the leaf sets of $T$; recall that these form a maximal slice partition of $G$. Then there is an $L_i$ such that $M$ is a module of $G[L_i]$, by lemma 4.2.1. Recall that $T_i$ is the MD tree for $G[L_i]$, by input condition 2 of algorithm 4.2.1. So by lemma 3.2.4, there are two cases to consider.

Assume first that there is a single node in $T_i$ corresponding to $M$. This node remains in $T'_i$ by definition of $T'$ being a refinement of $T$. Obviously, if $y \in L_j$, $j > i$, then $y$ is either universal to or isolated from $M$, by definition of modules. So by output condition 2 of algorithm 4.2.1 (and lemma 4.2.3), neither $u$ nor any of its descendants can be labelled dead.

Now assume that there is a single degenerate node $u$ in $T_i$ and $k' > 1$ of its children whose union corresponds to $M$. By the argument above, we can assume that all of these children remain in $T'_i$, and neither they nor any of their descendants are labelled dead. Moreover, since every $y \in L_j$, $j > i$, must be either universal to or isolated from $M$, it is easy to see that these children remain siblings in $T'_i$. So let $v$ be their shared parent in $T'_i$.

If $v$ has a child different from these $k'$ children, then it is easy to see that it must have been a child of $u$ in $T_i$. Let $c$ be such a child. Then, by definition, $c$ is a module of $G[L_i]$, and by remark 3.2.6, any union of $c$ and a subset of these $k'$ children is a module of $G[L_i]$. Moreover, since $c$ is a sibling in $T'_i$ of the $k'$ children corresponding to $M$, it is also easy to see that it must share the same neighbours as $M$ amongst the vertices in the $L_j$'s, $j > i$. Thus, $c$ is a module of $G$, possibly trivial. Furthermore, it follows that any union of the children of $v$ is a module of $G$. This contradicts $M$ being strong.

It now suffices to show that $v$ is not labelled dead. As already observed, the $k'$ children all have the same neighbours amongst the vertices in the $L_j$'s, $j > i$. Thus, if any of $v$'s children is ever marked, so too will be the other one. It follows that $v$ cannot be labelled dead.

We see in the proof of lemma 4.2.5 the role played by the nodes inserted into the $T_i$'s to refine them. They help isolate the strong modules of $G$ not containing $x$ that were not also strong modules of $G[P_i]$. We will see below that they also help produce a factorizing permutation of $G$. The labels left and right we assign them are designed for this purpose. Given a factorizing permutation, and the properties of it guaranteed by our recursion, it will be easy to identify the modules of $G$.
4.3  Factorizing Permutation

The output of algorithm 4.2.1 contains many nodes that do not correspond to modules of \( G \) (see lemma 4.2.4). These are unnecessary for the construction of the MD tree for \( G \) and can correspondingly be deleted. If we are careful in doing so, then we can produce a factorizing permutation in the sense of the following ordering:

**Definition 4.3.1.** Let \( T = T_1, \ldots, T_k \) be an ordered tree partition of \( G \), and let \( L_1, \ldots, L_k \) be the corresponding leaf sets. For each \( L_i \), we associate the ordering \( \sigma(L_i) \), defined as the order in which the leaves of \( T_i \) are encountered in an inorder traversal of \( T_i \). We use \( \sigma(T) \) to denote the ordering of the leaves in \( T \) defined by the ordering \( \sigma_1, \ldots, \sigma_k \).

That is, we will show that there is a way of deleting nodes in the output from algorithm 4.2.1 so that an ordered tree partition \( T' \) results, such that the ordering \( \sigma(T') \) is a factorizing permutation.

Now, by lemma 4.2.6, we already know that the strong modules of \( G \) not containing the pivot appear consecutively after algorithm 4.2.1. Moreover, that algorithm labelled the nodes it inserted as either *left* or *right*. The reason we did this was to help us rearrange nodes so that the strong modules containing the pivot also appear consecutively. This is what algorithm 4.3.1 below does. It uses the *left* and *right* signals to produce a factorizing permutation. An example of its operation is provided by figures 4.4 and 4.5.

We first show that algorithm 4.3.1 does not disrupt the previous consecutivity of the strong modules of \( G \) not containing the pivot:

**Lemma 4.3.2.** Let \( T = T_1, \ldots, T_k \) be the maximal slice tree partition of \( G \) input to algorithm 4.2.1, and let \( T' = T'_1, \ldots, T'_k \) be the maximal slice tree partition of \( G \) output by the algorithm. Suppose that \( T' \) is input to algorithm 4.3.1, and let \( T'' \) be the ordered tree partition of \( G \) output by the latter. If \( u \) is a node in \( T_i \) or \( T'_i \), then the leaves descendent from \( u \) appear consecutively in \( \sigma(T'') \).

**Proof.** No node in \( T_i \) is deleted by algorithm 4.2.1. During algorithm 4.3.1, only the roots of trees in the ordered tree partition input to the algorithm are deleted. And here, the subtrees rooted at the children of these nodes replace the tree in the ordering. Hence, either \( u \) exists in \( T'' \) or there is a consecutive sequence of trees in \( T'' \) the union of whose leaves correspond to those descendent from \( u \).

**Lemma 4.3.3.** Let \( T' \) be the ordered tree partition of \( G \) output by algorithm 4.3.1, with \( x \) the pivot. If \( M \) is a strong module not containing \( x \), then the vertices of \( M \) appear consecutively in \( \sigma(T') \).
Algorithm 4.3.1: Factorize($T$)

**Input:** A maximal slice tree partition $T = T_1, \ldots, T_k$ of some graph $G$, as produced by algorithm 4.2.1, where $T_1 = x$ is the pivot, and $L_1, \ldots, L_k$ are the corresponding leaf sets.

**Output:** An ordered tree partition $T'$ of $G$ such that if $M$ is a strong module, then $M - \{x\}$ appears consecutively in $\sigma(T')$. Moreover, the vertices in each co-component of $G[L_2]$, and each component of $G[L_i], i > 2$, also appear consecutively.

foreach $i = 1, \ldots, k$ do

foreach node $u$ in $T_i$ labelled as dead do

order the children of $u$ so that its left child appears before its right child;

label all of $u$’s ancestors as zombie;

endfch

foreach node $u$ in $T_i$ labelled zombie do

let $A$ be the children of $u$ that are labelled dead or zombie, and let $B$ be its other children;

if $i = 2$ then order the children of $u$ so that those in $A$ appear first;

else order the children of $u$ so that those in $A$ appear last;

endfch

initialize the ordered list of trees $T_i$ with $T_i$;

while there exists a tree $T$ in $T_i$ whose root is labelled dead or zombie do

replace $T$ in $T_i$ with the subtrees rooted at the children of its root, in the order fixed above;

endw

replace $T_i$ in $T$ with $T_i$;

endfch

return $T$;
Figure 4.4: The refined MD trees from figure 4.3 after nodes have been labelled \textit{zombie} and children have been reordered by algorithm 4.3.1: zombie nodes are indicated by cross-hatching; c.f. figure 4.3 to observe the reordering of children.

Figure 4.5: The trees of figure 4.4 after the \textit{dead} and \textit{zombie} nodes have been deleted by algorithm 4.3.1.
Proof. Immediate from lemmas 4.2.6 and 4.3.2.

Next, we summarize exactly how algorithm 4.3.1 made use of the left and right signals inserted by algorithm 4.2.1:

**Remark 4.3.4.** Let $T = T_1, \ldots, T_k$ be the maximal slice tree partition input to algorithm 4.3.1, and let $T'$ be the ordered tree partition output by algorithm 4.3.1. Let $u$ be a node in $T_i$. Then:

1. if $u$ is labelled dead, then the leaves descendant from its left child appear before those descendant from its right child in $\sigma(T')$;

2. if $u$ is not labelled dead, and it has children $v \neq v'$ such that $v$ or one of its descendants is labelled dead but neither $v'$ nor any of its descendants is labelled dead, then the leaves descendant from $v$ appear before those descendant from $v'$ in $\sigma(T')$.

Lastly, we use the next two results to simplify the problem of consecutively ordering the strong modules of $G$ containing the pivot:

**Lemma 4.3.5.** Let $P_1, \ldots, P_k$ be a maximal slice partition of $G$, and assume that $P_1 = \{x\}$ is the pivot. Let $M$ be a module containing $x$. If $C'$ is a co-component of $G[P_2]$, then either $C' \subseteq M$ or $C' \cap M = \emptyset$. Similarly, if $C$ is a component of $G[P_i], i > 2$, then either $C \subseteq M$ or $C \cap M = \emptyset$.

**Proof.** As $x \in M - C'$, we cannot have $M \subseteq C'$. So assume for contradiction that $C'$ and $M$ overlap. By remark 4.3.7, every vertex in $C' - M$ must be adjacent to every vertex in $C' \cap M$. This contradicts $C'$ being a co-component in $G[P_2]$. The argument for $C$ is similar.

**Remark 4.3.6.** Let $T = T_1, \ldots, T_k$ be the maximal slice tree partition of $G$ input to algorithm 4.2.1, and let $L_1, \ldots, L_k$ be the corresponding leaf sets.

1. If $C'$ is a co-component of $G[L_2]$, then either $C'$ corresponds to the root of $T_2$ or it corresponds to one child of the root;

2. if $C$ is a component of $G[L_i], i > 2$, then either $C$ corresponds to the root of $T_i$, or it corresponds to one child of the root.

The previous remark follows from the definition of MD trees. It suggests, along with lemmas 4.3.2 and 4.3.5, that the problem of consecutively ordering the strong modules of $G$ containing the pivot reduces to rearranging certain nodes in the trees input to algorithm 4.3.1. The algorithm does so as summarized in remark 4.3.4. The series of results below combine to prove that an ordering is produced in which the strong modules of $G$ containing the pivot do indeed appear consecutively. The first in this series follows easily from remark 4.1.7:

46
Remark 4.3.7. Let $P_1, \ldots, P_k$ be a maximal slice partition of $G$, where $P_1 = \{ x \}$ is the pivot, and let $M$ be a module containing $x$. If $C'$ is a co-component of $G[P_2]$, and $y \in C' - M$, then $y$ is universal to $M$. Similarly, if $C$ is a component of $G[P_i], i > 2$, and $y \in C - M$, then $y$ is isolated from $M$.

Lemma 4.3.8. Let $T = T_1, \ldots, T_k$ be the maximal slice tree partition of $G$ input to algorithm 4.2.1, and let $L_1, \ldots, L_k$ be the corresponding leaf sets. Suppose that the output of algorithm 4.2.1 is input to algorithm 4.3.1, and let $T'$ be the ordered tree partition of $G$ output by the latter. Assume that $x$ is the pivot of $T$. Let $M$ be a strong module containing $x$. Then the vertices in $L_2$ appear consecutively in $\sigma(T')$, say as the sequence $\sigma_2$, and the vertices in $M \cap L_2$ appear consecutively at the end of $\sigma_2$.

Proof. By lemma 4.3.2, the vertices in $L_2$ appear consecutively in $\sigma(T')$. So let $\sigma_2$ be the consecutive sequence in $\sigma(T')$ of vertices in $L_2$. By remark 4.3.6 and lemma 4.3.2, we know that if $C$ is a co-component of $G[L_2]$, then $C$ appears consecutively in $\sigma_2$. Thus, we can speak of one co-component of $G[L_2]$ appearing before another in $\sigma_2$.

Now, if $C$ is a co-component of $G[L_2]$, then either $C \subseteq M$ or $C \cap M = \emptyset$, by lemma 4.3.5. If all co-components of $G[L_2]$ are subsets of $M$ then we are done. So assume that there are co-components $C \neq C'$ such that $C \subseteq M$ and $C' \cap M = \emptyset$. Assume for contradiction that $C$ appears before $C'$ in $\sigma_2$.

Observe that $C$ and $C'$ correspond to a pair of sibling nodes in $T_2$, by definition of the MD tree and input condition 1 of algorithm 4.2.1. Thus, we will speak of $C$ and $C'$ equally as nodes. Let $T$ be the refinement of $T_2$ output by algorithm 4.2.1.

Consider the case where $C$ and $C'$ are not siblings in $T$. Consider the point during the execution of algorithm 4.2.1 in which they cease to be siblings. Let $u$ be their common parent before this point. Then there is a $y \in L_j, j > 2$, such that $y$ is universal to one but not the other. If $y \in M$, then $y$ must be universal to $C'$, by remark 4.3.7. On the other hand, if $y \notin M$, then $y$ cannot be universal to $C$, since $y$ is not adjacent to $x$, and thus $y$ is universal to $C'$. Thus, $C'$ will become a child of $u$'s left child, while $C$ will become a child of $u$'s right child. Therefore, in this case the desired contradiction follows by remark 4.3.4.

So assume that $C$ and $C'$ are siblings in $T$. Assume first that the parent of $C$ and $C'$ is labelled dead. The argument here is the same as above. So assume that the parent of $C$ and $C'$ is not labelled dead. Then there is no $y \in T_j, j > 2$, that is universal to one but not the other.

Now, every vertex in $M \cap L_i, i > 2$, is universal to $C'$, by remark 4.3.7. Therefore every vertex in $M \cap L_i, i > 2$, must also be universal to $C$. But then $(M - C) \cup C'$ is a module in $G$ that overlaps
Let the output of algorithm 4.2.1 be input to algorithm 4.3.1, and let $P$ be a module in $G$ that overlaps $M$, contradicting $M$ being strong. Thus, there is at least one $y \in L_i, i > 2$, that is adjacent to some vertex in $C'$. Therefore, either $u$ or one of its descendants must be labelled dead. The contradiction now follows from remark 4.3.4.

**Lemma 4.3.9.** Let $T = T_1, \ldots, T_k$ be the maximal slice tree partition of $G$ input to algorithm 4.2.1, and let $L_1, \ldots, L_k$ be the corresponding leaf sets. Suppose that the output of algorithm 4.2.1 is input to algorithm 4.3.1, and let $T'$ be the ordered tree partition of $G$ output by the latter. Assume that $x$ is the pivot of $T$. Let $M$ be a strong module containing $x$. Then the vertices in $L_i, i > 2$, appear consecutively in $\sigma(T')$, say as the sequence $\sigma_i$, and the vertices in $M \cap L_i$ appear consecutively at the beginning of $\sigma_i$.

**Proof.** Similar to the proof of lemma 4.3.8.

**Lemma 4.3.10.** Let $P_1, \ldots, P_k$ be a maximal slice partition of $G$, where $P_1 = \{x\}$ is the pivot. Let $M$ be a strong module containing $x$. If there exists a $P_j, j > 3$, such that $P_j \cap M \neq \emptyset$, then $P_2 \cap M \neq \emptyset$, and $P_3, \ldots, P_{j-1} \subseteq M$.

**Proof.** By remark 4.1.7, and condition 3(a) of remark 4.1.3, there is a vertex $y \in P_2$ that is universal to $P_3$ and isolated from $P_j$. If $y \notin M$, then $y$ must be universal to $M$, by remark 4.3.7. But as $P_j \cap M \neq \emptyset$, this is impossible. Thus, $y \in M$, meaning $P_2 \cap M \neq \emptyset$. On the other hand, if $z \in P_3$ is such that $z \notin M$, then $z$ is isolated from $M$, since $xz \notin E$. But we know that $y$ and $z$ are adjacent, and so $z \in M$. As $y$ is universal to $P_3$, we conclude that $P_3 \subseteq M$.

So assume that some $P_3, \ldots, P_i \subseteq M$, for some $3 \leq j - 1$. By condition 3(a) of remark 4.1.3, there is a vertex $w \in P_{\ell}, \ell < i + 1$, that is universal to $P_{i+1}$ but isolated from $P_j$. If $\ell = 2$, then as above, we must have $w \in M$. If $\ell > 2$, then $w \in M$ since $P_3, \ldots, P_i \subseteq M$. But of course, if there is a vertex in $P_{i+1} - M$, then that vertex must be isolated from $M$, since it is not adjacent to $x$. Thus, $P_{i+1} \subseteq M$. It follows by induction that $P_3, \ldots, P_{j-1} \subseteq M$.

**Lemma 4.3.11.** Let $T$ be the maximal slice tree partition of $G$ input to algorithm 4.2.1. Suppose that the output of algorithm 4.2.1 is input to algorithm 4.3.1, and let $T' = T'_1, \ldots, T'_k$ be the ordered tree partition of $G$ output by the latter. Assume that $x$ is the pivot of $T$. Let $M$ be a strong module containing $x$. Then the vertices in $M$ appear consecutively in $\sigma(T'_2, T'_1, T'_3, \ldots, T'_k)$.
Proof. The pivot must be the only leaf in \( T'_1 \). The result now follows by lemmas 4.3.8, 4.3.9, and 4.3.10.

\[ \text{Lemma 4.3.12. Algorithm 4.3.1 is correct.} \]

Proof. Immediate from lemmas 4.3.3 and 4.3.11.

The purpose of computing a factorizing permutation is to facilitate the identification of the strong modules of \( G \) containing the pivot. The task is further simplified by the special properties of the factorizing permutation computed by algorithm 4.3.1. In particular, we know that the (co-)components defining each such module (see lemma 4.3.5) will appear consecutively, by lemma 4.3.2 and remark 4.3.6. Below we reduce the problem of identifying the strong modules of \( G \) containing the pivot to one of determining their constituent (co-)components.

### 4.4 Modules Containing the Pivot

Based on the results of the previous section, we can view the factorizing permutation produced by algorithm 4.3.1 as one of the following form:

**Definition 4.4.1.** Let \( P_1, \ldots, P_k \) be a maximal slice partition of \( G \), with \( P_1 = \{x\} \) the pivot, and let \( C'_a, \ldots, C'_1, x, C_1, \ldots, C_b \) be an ordering of the (co-)components of the \( G[P_i] \)'s such that:

1. each \( C'_i \) is a co-component of \( G[P_2] \);
2. each \( C_i \) is a component of \( G[P_j] \), for some \( j > 2 \);
3. if \( 2 < i < j \), then the components of \( G[P_i] \) appear before those in \( G[P_j] \).

Assume that for each strong module \( M \) containing \( x \) there is a sequence \( C'_c, \ldots, C'_1, x, C_1, \ldots, C_d \) corresponding to \( M \). Then \( \sigma \) is called a pivot factorizing permutation of \( G \).

The definition is well-defined, by lemma 4.3.5. The pivot factorizing permutation for the example we have been following through the figures in this chapter is:

\[
\begin{align*}
C'_1 & = \{c, d\} & C_1 & = \{e\} \\
C'_2 & = \{b\} & C_2 & = \{y, z, f, g\} \\
C'_3 & = \{a\} & C_3 & = \{h, i, j, k\}
\end{align*}
\]

Let \( M \) be a strong module of \( G \) containing the pivot. We are interested in the problem of delineating the boundaries of \( M \) within a pivot factorizing permutation. Remark 4.3.7 will help us in this regard; it inspires the following two definitions:

49
Definition 4.4.2. Let $C'_a, \ldots, C'_1, x, C_1, \ldots, C_b$ be a pivot factorizing permutation of $G$, and let $P_1, \ldots, P_k$ be the underlying maximal slice partition. For each $y \in P_2$, let $\mu(y)$ be the smallest $j$ (possibly $j = 0$) such that every $z \in C_\ell, \ell > j$, is non-adjacent to $y$. For each $w \in P_i, i > 2$, let $\mu(w)$ be the smallest $j$ (possibly $j = 0$) such that every $z \in C'_\ell, \ell > j$, is adjacent to $w$. For each $C'_i$, we have $\mu(C'_i) = \max \{ \mu(y) \mid y \in C'_i \}$. For each $C_i$, we have $\mu(C_i) = \max \{ \mu(w) \mid w \in C_i \}$.

Definition 4.4.3. Let $C'_a, \ldots, C'_1, x, C_1, \ldots, C_b$ be a pivot factorizing permutation of $G$, and let $P_1, \ldots, P_k$ be the underlying maximal slice partition. For each $y \in P_i, i > 2$, let $\rho(y)$ be the largest $j > i$ such that there exists a $z \in P_j$ to whom $y$ is adjacent; if no such $j$ exists, then $\rho(y) = 0$. For each $C_i$, define $\rho(C_i) = \max \{ \rho(y) \mid y \in C_i \}$.

The table below illustrates the $\mu$ and $\rho$ values for the example we have been following through the figures in this chapter:

<table>
<thead>
<tr>
<th>(\mu)</th>
<th>(\mu)</th>
<th>(\rho)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(C'_1)</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>(C'_2)</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>(C'_3)</td>
<td>2</td>
<td>(\rho)</td>
</tr>
</tbody>
</table>

(4.2)

The role played by the previous definitions in defining the boundaries of $M$ is summarized by the next two lemmas:

Lemma 4.4.4. Let $C'_a, \ldots, C'_1, x, C_1, \ldots, C_b$ be a pivot factorizing permutation of $G$. Let $M' \subset M$ be two strong modules containing $x$, such that they correspond to two consecutive nodes on the path from $x$ to the root in the MD tree for $G$. Then:

1. $M$ is series if and only if there is no $C_i$ such that $C_i \cap (M - M') \neq \emptyset$;

2. $M$ is parallel if and only if there is no $C'_i$ such that $C'_i \cap (M - M') \neq \emptyset$, and there is no $C_i$ such that $C_i \cap M - M' \neq \emptyset$ and $\rho(C_i) > 0$.

Proof. Assume first that $M$ is series, and suppose for contradiction that there is some $C_i$ such that $C_i \cap M - M' \neq \emptyset$. By lemma 4.3.5, we must have $C_i \subseteq M - M'$. There can be no edges between vertices in $C_i$ and those in $M'$, by remark 4.3.7. It follows that $C_i$ and $M'$ must be part of the same co-component in $G[M]$. But this is impossible as $M'$ is a child of $M$ in the MD tree for $G$, and thus $M'$ alone (without any vertex in $C_i$) is a co-component in $G[M]$.

Now assume that there is no $C_i$ such that $C_i \cap M - M' \neq \emptyset$. We must have $M - M' \neq \emptyset$, so given the pivot factorizing permutation, we have $M = M' \cup C'_i \cup \cdots \cup C'_j$, for some pair $i, j$, by
lemma 4.3.5. Also, \( C_i' \) is universal to \( M' \), by remark 4.3.7. Of course, these \( C_i' \)'s are all universal to each other. Hence, \( G[M] \) is disconnected, meaning \( M \) is series.

Suppose that \( M \) is parallel. The proof showing that there is no \( C_i' \) such that \( C_i' \cap M - M' \neq \emptyset \) is similar to the one above for series modules and there being no \( C_i \) such that \( C_i \cap M - M' \neq \emptyset \). We instead focus on showing that there is no \( C_i \) such that \( C_i \cap M - M' \neq \emptyset \) and \( \rho(C_i) > 0 \). Assume for contradiction that such a \( C_i \) exists, and without loss of generality, choose \( i \) to be the smallest index for which this is true. Let \( \rho(C_i) = j \). By lemmas 4.3.5 and 4.3.11, we must have \( C_j \subseteq M \).

Let \( P_1, \ldots, P_k \) be the maximal slice partition underlying the pivot factorizing permutation of the lemma. Assume that \( C_i \subseteq P_i \) and \( C_j \subseteq P_r \). By condition 3(a) of remark 4.1.3, we know that there is an \( \ell' < \ell \), and a vertex \( y \in P_r \), such that \( y \) is universal to \( P_i \) and isolated from \( P_r \). We cannot have \( y \in M' \), because \( C_i \) must be isolated from \( M' \), by remark 4.3.7. Moreover, we cannot have \( y \notin M \), since \( C_i, C_j \subseteq M \). Thus, \( y \in M - M' \). But since this is a pivot factorizing permutation, every component appearing before \( C_i \) must also be in \( M \). Thus, by choice of \( i \), we must have \( y \in (M - M') \cap C_d' \), for some \( d \). The contradiction follows since we just showed that there can be no \( C_d' \) such that \( C_d' \cap M - M' \neq \emptyset \).

Now suppose that there is no \( C_i' \) such that \( C_i' \cap M - M' \neq \emptyset \), and there is no \( C_i \) such that \( C_i \cap M - M' \neq \emptyset \) and \( \rho(C_i) > 0 \). By the latter fact, there are no edges between \( C_i \)'s in \( M - M' \). Of course, no vertex in \( C_i \) can be adjacent to a vertex in \( M' \), by remark 4.3.7. Thus, \( G[M] \) must be disconnected, meaning \( M \) is parallel.

\[ \Box \]

**Lemma 4.4.5.** Let \( C_{a'}, \ldots, C_{i'}, x, C_1, \ldots, C_b \) be a pivot factorizing permutation of \( G \). Let \( M' \subseteq M \) be two strong modules containing \( x \), such that they correspond to two consecutive nodes on the path from \( x \) to the root in the MD tree for \( G \). Let \( \ell \) be the largest index such that \( C_{i'} \cap M' \neq \emptyset \), and let \( r \) be the largest index such that \( C_{r'} \cap M' \neq \emptyset \). If \( M \) is prime, then:

1. \( \ell \neq a \) and \( r \neq b \);
2. \( \mu(C_{\ell+1}') > r \);
3. either \( \mu(C_{r+1}') > \ell \) or \( \rho(C_{r+1}) > 0 \);
4. \( C_{\ell+1}' \cap M \neq \emptyset \) and \( C_{r+1} \cap M \neq \emptyset \).

**Proof.** Since \( M \) is prime, we cannot have \( \ell \neq a \) and \( r = b \), by condition 1 of lemma 4.4.4. Similarly, by condition 2 of that lemma, we cannot have \( \ell = a \) and \( r \neq b \). Moreover, we cannot have \( \ell = a \) and \( r = b \), since otherwise \( M \) would be empty. Thus, \( \ell \neq a \) and \( r \neq b \). By a similar argument, we must have \( C_{\ell+1}' \cap M \neq \emptyset \) and \( C_{r+1} \cap M \neq \emptyset \).
Figure 4.6: The spine built by algorithm 4.4.1 for the pivot factorizing permutation from equation (4.1), based on the $\mu$- and $\rho$-values from equation (4.2).

Suppose for contradiction that $\mu(C'_{\ell+1}) \leq r$. Then $M'' = M' \cup C'_{\ell+1}$ is a module in $G$. We just showed that $C'_{\ell+1} \cap M \neq \emptyset$, so we must have $C'_{\ell+1} \subseteq M$, by lemma 4.3.5; hence, $M'' \subseteq M$. In fact, $M'' \subset M$, since we also just showed that $C_{r+1} \cap M \neq \emptyset$. Obviously, $M' \subset M''$. Therefore, $M'' \neq M$ and $M'' \neq M'$, meaning $M''$ is not a strong module, by choice of $M$ and $M'$. But lemma 3.2.4 now implies that $M$ is degenerate, a contradiction. Thus, $\mu(C'_{\ell+1}) > r$. A similar argument shows that either $\mu(C_{r+1}) > \ell$ or $\rho(C_{r+1}) > 0$.

It is now a straightforward task to identify the strong modules of $G$ containing the pivot. Algorithm 4.4.1 below does so merely by implementing lemmas 4.4.4 and 4.4.5. Its name $BuildSpine$ reflects its output, the “spine” of the MD tree for $G$; namely, the path from the pivot to the root in the MD tree for $G$. An example of its operation is provided by figure 4.6.

Lemma 4.4.6. Algorithm 4.4.1 is correct.

Proof. Assume as an induction hypothesis that, after $i \geq 0$ number of iterations of the outermost while loop, the indices $\ell$ and $r$ are such that $C'_{\ell}, \ldots, C'_{1}, x, C_{1}, \ldots, C_{r}$ corresponds to a strong module $M_{i}$ containing $x$. Let $M_{i+1}$ be the smallest strong module properly containing $M_{i}$.

Assume first that $M_{i+1}$ is series. Then by lemma 4.4.4, there is no $C_{j}$ such that $C_{j} \cap M_{i+1} - M_{i} \neq \emptyset$. But obviously, $M_{i+1} - M_{i} \neq \emptyset$, and so there must be some $C'_{j}$ such that $C'_{j} \cap M_{i+1} - M_{i} \neq \emptyset$. Thus, $C'_{j} \subseteq M_{i+1} - M_{i}$, by lemma 4.3.5. Hence, $\mu(C'_{j}) \leq r$. Amongst all such $j$, choose the largest. Then $\mu(C'_{j+1}) > r$, meaning $C'_{j+1} \cap M_{i+1} = \emptyset$. Therefore, since this is a pivot factorizing permutation, we must have $M_{i+1} = M_{i} \cup C'_{\ell+1} \cup \cdots \cup C'_{j}$, where $j \geq \ell+1$. But of course, in this case, in algorithm 4.4.1 we end up with $M = C'_{\ell+1} \cup \cdots \cup C'_{j}$, and therefore $M_{i+1}$ is correctly identified. The argument for the parallel case is similar.
Algorithm 4.4.1: BuildSpine(σ)

**Input:** A pivot factorizing permutation $σ = C'_a, \ldots, C'_1, x, C_1, \ldots, C_b$ for which $μ(C'_i)$ has been computed for each $C'_i$, and for which $μ(C_i)$ and $ρ(C_i)$ have been computed for each $C_i$.

**Output:** A rooted tree $T$ whose nodes correspond to the strong modules containing $x$, each one properly labelled by the module’s type, where the leaves of $T$ are the elements of $σ$, and each node and leaf is descendent from all strong modules containing it.

$T ← x$;  
$ℓ, r ← 0$; // left and right indexes

while $ℓ ≠ a$ or $r ≠ b$ do

    $M ← ∅$; // The strong module containing $x$ that will be identified this iteration.

    // Locating series modules
    $ℓ ← ℓ + 1$;
    while $μ(C'_ℓ) = r$ and $ℓ ≤ a$ do $M ← M \cup \{C'_ℓ\}; ℓ ← ℓ + 1$;
    $ℓ ← ℓ - 1$;

    // Locating parallel modules
    if $M = ∅$ then
        $r ← r + 1$;
        while $μ(C_r) = ℓ$ and $ρ(C_r) = 0$ and $r ≤ b$ do $M ← M \cup \{C_r\}; r ← r + 1$;
        $r ← r - 1$;
    endif

    // Locating prime modules
    if $M = ∅$ then
        $ℓ ← ℓ + 1$; $ℓ' ← ℓ$;
        $r ← r + 1$; $r' ← r$;
        while $μ(C_{ℓ'}) > ℓ$ or $ρ(C_{ℓ'}) > r$ or $μ(C'_{r'}) > r$ do
            $t ← μ(C_{ℓ'})$;
            $m ← max\{μ(C'_{ℓ'}), ρ(C_{ℓ'})\}$;
            $M ← M \cup C'_{ℓ'} \cup \cdots \cup C'_t \cup C_t \cup \cdots \cup C_m$;
            $ℓ' ← max\{μ(C'_i) \mid ℓ < i ≤ t\}$;
            $r' ← max\{μ(C_i) \mid r < i ≤ m\} \cup \{ρ(C_i) \mid r < i ≤ m\}$;
            $ℓ ← ℓ'$;
            $r ← r'$;
        endw
    endif

    create a new node $u$ with the elements of $M$ as its children;
    label $u$ as series if there is no $C_i ∈ M$, as parallel if there is no $C'_i ∈ M$, otherwise as prime;
    update $T$ by making $u$ the parent of $T$’s root;
endw

return $T$;
So assume that \( M_{i+1} \) is prime. Then by conditions 1-3 of lemma 4.4.5, neither of the first two inner while loops execute. This means the third inner while loop must execute. Now, by condition 4 of lemma 4.4.5, we know that \( C'_{\ell+1} \subseteq M_{i+1} \) and \( C_{r+1} \subseteq M_{i+1} \). Given this, it is an easy inductive argument (based on the definition of modules and the definition of \( \mu \) and \( \rho \)) to show that each (co-)component added to \( M \) during the third inner while loop belongs to \( M_{i+1} \). Thus, after the loop terminates, \( M \cup M_i \subseteq M_{i+1} \). But by the loop’s exit conditions, we know that \( M \cup M_i \) is a module.

Now, by lemma 3.2.4, we have two cases for the module \( M \cup M_i \). In the first, \( M \cup M_i \) corresponds to a node in the MD tree for \( G \), and is therefore a strong module. Since \( M \neq \emptyset \), this strong module must be an ancestor of \( M_i \). But recall that \( M \cup M_i \subseteq M_{i+1} \). So by choice of \( M_{i+1} \), we must have \( M_{i+1} = M \cup M_i \) in this case.

In the second case, \( M \cup M_i \) corresponds to the union of \( k > 1 \) children of a degenerate node in the MD tree for \( G \). Since \( M \neq \emptyset \), this node must be an ancestor of \( M_i \); and by choice of \( M_{i+1} \), this node must be an ancestor of \( M_{i+1} \). But recall that \( M \cup M_i \subseteq M_{i+1} \), which contradicts \( k > 1 \).

Algorithm 4.4.1 computes the strong modules of \( G \) containing the pivot. Those not containing the pivot are computed by algorithm 4.2.1, as in lemma 4.2.6. The next section explains how the two are assembled to form the MD tree for \( G \).

4.5 Conquering

To this point, we have independently computed the strong modules of \( G \) containing the pivot (lemma 4.4.6) and those not containing the pivot (lemma 4.2.6). The process has been simple in both instances; we will eventually see below that the process of combining them to construct the MD tree for \( G \) is no more difficult. However, we need to be careful while doing so to ensure the efficiency of the algorithm. The next two remarks are designed to address this concern; both follow easily from the definitions involved:

**Remark 4.5.1.** Let \( P_1, \ldots, P_k \) be a maximal slice partition of \( G \). Then \( G \) is disconnected if and only if there is no \( y \in P_k \) with an incident active edge. Moreover, every vertex \( y \in P_i, i < k \), has an incident active edge; and when \( G \) is disconnected, \( P_1 \cup \cdots \cup P_{k-1} \) is a component of \( G \).

**Remark 4.5.2.** Assume that \( G \) is disconnected, and let \( C \) be one of its components. Moreover, let \( T_c \) be the MD tree for \( G[C] \), and let \( T_d \) be the MD tree for \( G[V(G) - C] \). Then the MD tree for \( G \) can be formed as follows:
• if the root of $T_d$, call it $u$, is parallel, then make the root of $T_c$, call it $v$, a child of $u$, and if $v$ is also parallel, replace it by its children;

• otherwise, create a new parallel node whose children are $u$ and $v$.

The application of the previous two remarks is as follows. Let $P_1, \ldots, P_k$ be a maximal slice partition of $G$, and assume that $G$ is disconnected. In this case we will have recursively computed the MD tree for $G[P_k]$. Thus, it suffices to compute the MD tree for the component defined by $C = P_1 \cup \cdots \cup P_{k-1}$. Once we have done that, the cost to assemble the MD tree for $G$ is proportional to the number of children in the MD tree for $G[C]$. The full importance of this will become clear when we come to discuss the running-time of our algorithm. For now, we only note that each vertex in $P_i$, $i < k$, has an incident active edge, while those in $P_k$ do not. That will prove vital; not only in assembling the MD tree for $G$, but also in the construction of the MD tree for $G[C]$. The latter can obviously be performed as in the remark below:

**Remark 4.5.3.** Let $x$ be a vertex of some connected graph $G$, and let $M_1 \subset \cdots \subset M_k$ be the strong modules containing $x$ (where $M_1 = \{x\}$). Then the MD tree for $G$ can be obtained as follows:

• create a rooted tree $T$ whose vertices are the $M_i$’s, such that $M_i$ is the parent of $M_{i-1}$;

• for each strong module $M$ not containing $x$, make the root of the MD tree for $G[M]$ a child in $T$ of the smallest $M_i$ such that $M \subseteq M_i$;

• for each degenerate node $u$ in $T$ whose parent has the same type, replace $u$ by its children.

Let $M$ be a strong module of $G$ not containing the pivot. Then $M$ is computed by algorithm 4.2.1, as in lemma 4.2.6. Now, not only does algorithm 4.4.1 compute the strong modules of $G$ containing the pivot, it also computes, for each (co-)component in the pivot factorizing permutation, which of them is the smallest to properly contain that (co-)component. By definition, no strong module of $G$ not containing the pivot can overlap one containing the pivot. Hence, to determine which node on the spine to make $M$ a child, in the sense of remark 4.5.3, it suffices to determine the (co-)component(s) it intersects.

The preceding discussion gives rise to algorithm 4.5.1 below. Recall its appearance in algorithm 4.1.1 of section 4.1. The results presented in the sections that followed culminate below in its specification. An example of its operation is provided by figure 4.7.

**Lemma 4.5.4.** Algorithm 4.5.1 is correct.

*Proof.* Immediate from lemmas 4.2.3, 4.2.6, 4.3.12, and 4.4.6, and remarks 4.5.3 and 4.5.2. \qed
Algorithm 4.5.1: ConquerMDTree(T)

**Input:** A maximal slice tree partition \( T = T_1, \ldots, T_k \) of some graph \( G \), such that if

\[
L_1, \ldots, L_k
\]
are the corresponding leaf sets, then \( T_i \) is the MD tree for \( G[L_i] \).

Moreover, we assume that each leaf \( x \in L_i \) has an associated set \( \alpha(x) \) consisting of its neighbours amongst the leaves of the \( T_j \)'s, \( j < i \).

**Output:** The MD tree for \( G \).

let \( x \) be the pivot of \( T \);

if there is no \( y \in L_k \) such that \( |\alpha(y)| > 0 \) then \( k' \leftarrow k - 1 \); else \( k' \leftarrow k \);

foreach \( i = 1, \ldots, k' \) do

if \( i = 2 \) then label the leaves in \( L_i \) by their co-components in \( G[L_i] \);

else label the leaves in \( L_i \) by their components in \( G[L_i] \);

// to identify the pivot factorizing permutation later

endfch

\( T' \leftarrow TreeRefinement(T_1, \ldots, T_{k'}) \); // Algorithm 4.2.1

\( T'' \leftarrow Factorize(T') \); // Algorithm 4.3.1

let \( \sigma = C'_a, \ldots, C'_1, x, C_1, \ldots, C_b \) be the pivot factorizing permutation defined by \( T'' \);

foreach \( i = 1, \ldots, k' \) do compute \( \mu(y) \) for each \( y \in L_i \);

compute \( \mu(C'_i), \mu(C_i), \) and \( \rho(C_i) \) for each (co-)component in \( \sigma \);

\( T \leftarrow BuildSpine(\sigma) \); // Algorithm 4.4.1

replace the leaves of \( T \) with the corresponding trees in \( T'' \);

if \( k' = k - 1 \) then

if \( T_k' \)'s root is parallel then update \( T \) by making the root of \( T_k' \) the parent of \( T \)'s root;

else update \( T \) by adding a new parallel node as the parent of \( T \)'s root, and adding the root of \( T'_k \) as a child of this new root;

endif

foreach degenerate node \( u \) in \( T \) whose parent has the same type do

replace \( u \) by its children;

endfch

return \( T \);
As promised in section 4.1, our modular decomposition algorithm can be specified as we do in algorithm 4.5.2.

**Algorithm 4.5.2: MDTree(G)**

*Input:* A graph $G$.

*Output:* The MD tree for $G$.

$$(T, \mathcal{P}) \leftarrow \text{DivideMDTree}(V(G), \emptyset);$$

*return* $T$;

**Lemma 4.5.5.** Algorithm 4.5.2 is correct.

*Proof.* Immediate from lemmas 4.1.5 and 4.5.4. \hfill \Box

The simplicity of algorithm 4.5.2 is manifest in its constituent parts; even despite the detail in which each is presented. We present an implementation below to prove the efficiency of the algorithm. We will see there that whatever remains to be specified in this regard amounts to no more than simple tree traversals.

### 4.6 Implementation and Running Time

The implementation of our modular decomposition algorithm will be as straightforward as its statement has been. Let $G$ be the graph input to algorithm 4.5.2, and assume that $G$ has $n$ vertices
and \( m \) edges. We require that \( G \) is input as an adjacency-list. We will implement the \( \alpha \)-lists as a second linked-list pointed to by each vertex in this representation, in addition to the list of their neighbours. Aside from these, the only data-structure required by our modular decomposition algorithm will be an ordered list of trees.

Algorithm 4.5.2 indirectly manipulates ordered lists of trees through its calls to algorithms 4.2.1 and 4.3.1. The roots of each tree will be maintained in a doubly-linked list; the trees will be stored in the standard way: siblings in a doubly-linked list, each vertex having a pointer to its parent; and each root will maintain a pointer to one of its leaves. The other structures processed during the execution of algorithm 4.5.2 are ordered partitions – through calls to algorithm 4.1.1 – and ordered lists of (co-)components – through calls to algorithm 4.4.1. However, both of these can be viewed as ordered lists of trees (of depth 1 and depth 0, respectively), and will be represented accordingly.

We will show below that only simple traversals will be required of the \( \alpha \)-lists and ordered lists of trees. To prove the efficiency of these traversals, we will need remark 4.5.1 and the observations below:

**Remark 4.6.1.** During the execution of algorithm 4.5.2, each vertex is a pivot precisely once, and each edge becomes active precisely once.

**Remark 4.6.2.** Let \( T \) be a rooted tree and let \( L \) be the set of its leaves. If every node in \( T \) has at least two children, then \( \text{\#}(T) \leq 2 \cdot |L| \).

**Remark 4.6.3.** Every tree processed by algorithms 4.2.1 or 4.3.1 is such that every one of its nodes has at least two children.

Remark 4.6.1 follows easily from the definitions involved. To see remark 4.6.2, start by placing two tokens on every leaf in \( T \). Then recursively slide a single token from every vertex in \( T \) to its parent. Obviously, each vertex in \( T \) will be covered by a token and have one free to slide up.

Finally, for remark 4.6.3, observe that every node in an MD tree has two children, by remark 3.2.7. These are the trees input to algorithm 4.2.1. Nodes are not deleted by algorithm 4.2.1, and when a node is created, it is assigned distinct left and right children. On the other hand, trees output by algorithm 4.2.1 are those input to algorithm 4.3.1. Nodes are not created by the latter, and when nodes are deleted, they are replaced by their children.

Below we revisit the sections of this chapter in turn, specifying an implementation for the algorithms presented in each, and applying the remarks noted above to demonstrate their contribution to the running-time of algorithm 4.5.2.
4.6.1 Dividing by LBFS

As in section 4.1, we will concentrate on the portion of algorithm 4.1.1 other than its call to algorithm 4.5.1 (preferring to undertake its analysis in parts that follow below). The remainder of algorithm 4.1.1 will be divided into two parts: the while loop performing the recursion, and all that precedes it.

What precedes the while loop in algorithm 4.1.1 amounts to a simple application of partition refinement. To split a class $P$ into $P \cap D$ and $P - D$ given our data-structure, we merely need to traverse the elements of $D$ twice: once to group them, and once to remove them to form $P \cap D$; what remains of $P$ represents $P - D$. A constant time operation then initiates the pointer $P \cap D$ and $P - D$ must maintain to one of its leaves. The cost of this is $O(|D|)$.

In the case of algorithm 4.1.1, we have $D = N(x)$. As we traverse the elements in $N(x)$, we can append $x$ to $\alpha(y)$, for each $y \in N(x) \cap (S \cup S')$, as required. The total cost is $O(|N(x)|)$. Now, $x$ becomes the pivot in the call to algorithm 4.5.1 after the while loop. But by remark 4.6.1, each vertex is pivot exactly once during the execution of algorithm 4.5.2. As such, the portion of algorithm 4.1.1 preceding its while loop contributes $O(n + m)$ total cost to the execution of algorithm 4.5.2.

Let $P_1, \ldots, P_{k'}$ be the sequence of classes removed from $\mathcal{P}$ during the while loop of algorithm 4.1.1. Not counting the work performed during the recursive calls, the loop takes time $O(k')$ to execute. Now, $\{x\}, P_1, \ldots, P_{k'}$ is a maximal slice partition of $G[S]$. Thus, every $P_i$ but possibly $P_{k'}$ contains a vertex with an incident active edge, by remark 4.5.1. But by remark 4.6.1, each edge is active precisely once during the execution of algorithm 4.5.2. Thus, the while loop in algorithm 4.1.1 contributes $O(m)$ total cost to the execution of algorithm 4.5.2.

4.6.2 Modules Not Containing the Pivot

Consider the process of marking nodes, as in algorithm 4.2.1. We propose to implement this by first marking all vertices/leaves in $\alpha(y)$ by scanning its elements once. Once these have been identified, a simple recursive, bottom-up marking scheme marks nodes $u$ whose children are all marked. The cost is clearly proportional to the number of nodes thus marked. But every node $u$ thus marked is such that every node in the subtree rooted at $u$ is also marked. Hence, by remark 4.6.3, the cost of marking all such nodes is $O(|\alpha(y)|)$. Of course, the cost of clearing these marks is the same.

Once marked, the creation of new children corresponding to the $A$ and $B$ in the algorithm can be effected as described above for partition refinement: a single pass through $\alpha(y)$ groups the children in $A$; a second pass removes them as children; what remains is a node corresponding to $B$. The
replacement now occurs in the obvious way by creating a new node to replace that corresponding to \( B \), making \( A \) and \( B \) this new node’s children. This replacement and the subsequent marking and labelling required by the algorithm are constant time operations. In this case, neither new child can be the root of a tree, so no pointer need be initialized to a descendant leaf. The cost of the replacement is therefore \( O(|\alpha(y)|) \).

Each leaf \( y \) of a tree input to algorithm 4.2.1 is processed once by the algorithm. As observed above, it costs \( O(|\alpha(y)|) \) to process each such leaf. But notice that algorithm 4.2.1 is only ever called during the execution of algorithm 4.5.2 by algorithm 4.5.1. The latter ensures that every leaf input to algorithm 4.2.1 will have at least one incident active edge, by remark 4.5.1. Of course, each \( z \in \alpha(y) \) corresponds to an active edge incident to \( y \). Thus, by remark 4.6.1, algorithm 4.2.1 contributes \( O(n + m) \) total cost to the execution of algorithm 4.5.2.

### 4.6.3 Factorizing Permutation

Algorithm 4.3.1 consists of four loops: an outer one, and three inner ones. The first inner loop can be implemented by a bottom-up scan of the trees in the ordered list: as a left child (whose parent is dead) is encountered, it is placed to the left of its only other sibling; as a right child (whose parent is dead) is encountered, it is placed to the right of its only other sibling. Similarly for the second inner while loop: as a dead or zombie child (whose parent is zombie) is encountered, it is placed either before or after all other siblings. Finally, the third inner loop can obviously be implemented by a top-down scan of each tree in the ordered list. A final top-down scan of the ordered list of trees can initialize the pointer each root must maintain to one of its leaves.

The total cost of all these scans is clearly proportional to the combined size of the trees in the ordered list. This is proportional to the number of leaves input to algorithm 4.3.1, by remark 4.6.3. Now, algorithm 4.3.1 is only called during algorithm 4.5.2 by algorithm 4.5.1. The latter ensures that every vertex in each (co-)component has at least one incident active edge. So by remark 4.6.1, the contribution of algorithm 4.3.1 to the total cost of algorithm 4.5.2 is \( O(m) \).

### 4.6.4 Modules Containing the Pivot

Let \( C'_a, \ldots, C'_1, x, C_1, \ldots, C_b \) be the pivot factorizing permutation input to algorithm 4.4.1. It is obvious upon examination of the algorithm that it runs in time \( O(a + b) \). As no \( C'_i \) or \( C_i \) is empty, \( a + b \) is proportional to the total number of vertices in these (co-)components. Now, algorithm 4.4.1 is only ever called during the execution of algorithm 4.5.2 by algorithm 4.5.1. The latter ensures that every vertex in each (co-)component has at least one incident active edge, by remark 4.5.1.
Hence, the contribution of algorithm 4.4.1 to the total cost of algorithm 4.5.2 is $O(m)$.

### 4.6.5 Conquering

Having already discussed the implementation of algorithms 4.2.1, 4.3.1, and 4.4.1, as well as the cost they contribute to the running-time of algorithm 4.5.2, we ignore the calls made to them in algorithm 4.5.1. The rest of the algorithm amounts to preparation for each of these calls; we focus on that cost here.

Identifying the pivot of $T$ is a constant time operation since, by definition, it must be the only vertex of $T_1$. By condition 2 of remark 4.1.3, if $y, z \in L(k)$, then $\alpha(y) = \alpha(z)$. Thus, the cost to determine $k'$ is $O(k)$, since we need only traverse $T$ to get to $T_k$, follow the pointer from its root to one of its leaves, and then check if $\alpha(y)$ is empty. The total cost of identifying the pivot and $k'$ is therefore $O(k)$.

By definition, the (co-)components of the $G[L_i]$’s correspond either to the root of $T_i$ or the children of the root of $T_i$, since each $T_i$ is the MD tree for $G[L_i]$. Hence, the leaves of the $T_i$’s can be labelled by their (co-)components with a simple top-down scan of each tree. The total cost is clearly proportional to the total number of the nodes in the $T_i$’s (up to $k'$).

Having previously labelled each leaf $y \in L_i$ by their (co-)component in $G[L_i]$, a simple scan through these leaves computes the pivot factorizing permutation $\sigma$. The total cost is clearly proportional to the total number of leaves in the $T_i$’s (up to $k'$).

To compute the $\mu$- and $\rho$-values for each leaf, we need to introduce a temporary $\alpha'$ list for each leaf. These will be defined such that if $y \in L_i$, then $\alpha'(y)$ consists of $y$’s neighbours in the $T_j$’s, $j > i$. They can be computed by a single scan of the $\alpha$-lists for each leaf: as $z$ is encountered scanning $\alpha(y)$, then $y$ is appended to $\alpha'(z)$.

By processing each leaf $z$ in order, from left to right in $\sigma(T'')$, and scanning $\alpha'(z)$ as it is encountered, then $\mu(y)$ can easily be computed in the obvious way for each $y \in C_i$. If we scan the leaves instead from right to left, then $\rho(y)$ can be computed in the obvious way for each $y \in C_i$, and $\mu(w)$ can be computed in the obvious way for each $w \in C_i$. Once these are computed, another scan can compute $\mu(C_i'), \mu(C_i)$, and $\rho(C_i)$ in the obvious way. A final scan can delete these temporary $\alpha'$-lists. The total cost is clearly proportional to the total number of leaves in $T''$.

The rest of the work to define $T$ can clearly be completed in the obvious way by further traversals of the trees in $T''$ and constant time pointer assignments; this includes the initialization of the pointer from the root of $T$ to one of its leaves. The cost is clearly proportional to the total number of nodes in $T''$.

Now, the total number of nodes in $T''$ is proportional to the total number of its leaves, by
remark 4.6.3. These are also the leaves of the $T_i$’s (up to $k'$). Each of these leaves has an incident active edge, by remark 4.5.1. Therefore, by remark 4.6.1, the contribution of the portion of algorithm 4.5.1 not executed in calls to other algorithms is $O(m)$.

Finally, we conclude the following by combining the results presented in sections 4.6.1-4.6.5:

**Theorem 4.6.4.** Algorithm 4.5.2 can be implemented to run in time $O(n + m)$.

In one sense, the previous theorem is unsurprising. It is already well-known that modular decomposition can be computed in linear-time: as noted in chapter 1, there are three algorithms for that purpose [102, 47, 56]. Rather, theorem 4.6.4 is remarkable for confirming that it can be done so simply and practically. Conceptually, algorithm 4.5.2 follows merely from lemma 3.2.4 and basic definitions of modules and their types. Its implementation requires no more than basic traversals of lists and trees. The interested reader can see a full implementation of a preliminary version of this algorithm (see [130] for this version) at [128].

The key to the algorithm’s simplicity has been unifying two previous approaches to the problem – divide and conquer and factorizing permutations – and integrating them with LBFS. One important point is that our algorithm cannot use existing algorithms that compute the MD tree from a factorizing permutation (see chapter 2 for a description). These all run in time $\Omega(n + m)$, whereas we must be able to compute the MD tree from a factorizing permutation in time $O(n)$. We manage this by computing a special factorizing permutation – one in which the (co-)components are consecutive. This is done by refining the recursively computed MD trees of each slice in a kind of retroactive tie-breaking scheme. We will discuss other possible applications of this retroactive tie-breaking scheme in the conclusion (chapter 7).
Chapter 5

Simpler and Efficient Split Decomposition

Responding to the difficulty of the only linear-time split decomposition algorithm [55], research has focused on presenting simpler alternatives that do not sacrifice its efficiency. This chapter presents a simpler alternative that is nearly as efficient. We define an incremental characterization of split decomposition in terms of the split-tree (see chapter 3). Adding vertices according to an LBFS ordering allows for the efficient implementation of this characterization. The implementation uses union-find, which introduces an inverse Ackermann factor into the running-time. It is only this that prevents us from achieving linear-time.

We begin by considering the problem of inserting a single vertex into an existing split-tree, and then extend that case in the natural way. We will prove that LBFS orderings restrict the appearance of twins and stars in the split-tree. An implementation based on these restrictions is then proposed. The proof of the running-time requires a detailed and technical charging argument that is the primary difficulty associated with the algorithm. Despite this, and despite the algorithm not running in linear-time, we are still able to extend it to improve the running-time of circle graph recognition. The next chapter builds on the algorithm presented here to develop the first sub-quadratic circle graph recognition algorithm.

This chapter is based on the technical report [79], jointly developed with Derek Corneil, Emeric Gioan, and Christophe Paul.
5.1 Incremental Split-Decomposition

Consider $ST(G)$ and suppose that $x \notin V(G)$. This section derives an algorithm to compute $ST(G + (x, S))$ from $ST(G)$, where $S \subseteq V(G)$. We will assume that $G$ is connected, and that $S \neq \emptyset$; thus, $G + (x, S)$ is also connected.

The simplest case is when $|S| = 1$, meaning $x$ is a pendant in $G + (x, S)$. In this case, $ST(G + (x, \{y\}))$, where $x \notin V(G)$.

The changes to $ST(G)$ are entirely determined by states assigned to marker vertices:

**Definition 5.1.1.** Let $q$ be a marker vertex in a GLT $(T, \mathcal{F})$, and let $S$ be a subset of $T$’s leaves. The state of $q$ with respect to $S$ is:

- perfect if $S \cap L(q) = A(q)$;
- empty if $S \cap L(q) = \emptyset$;
- mixed otherwise.

For each node $u$, we let $P(u) = \{q \in V(u) \mid q \text{ perfect}\}$; $M(u) = \{q \in V(u) \mid q \text{ mixed}\}$; $E(u) = \{q \in V(u) \mid q \text{ empty}\}$; and $NE(u) = V(u) - E(u)$.\(^1\) For simplicity, we may not always use $NE$ for “not empty”.

---

\(^1\)“NE” for “not empty”.

64
mention $S$ when discussing states, instead leaving its existence implied.

We will use these states to characterize how $ST(G)$ must change to form $ST(G + (x, S))$. We therefore emphasize the following basic properties, which will form the foundation of the proofs to come:

**Remark 5.1.2.** If $q$ is opposite a leaf, then $q$ is either perfect or empty.

**Remark 5.1.3.** Let $(T, \mathcal{F})$ be a GLT, let $u$ and $v$ be adjacent nodes in $T$, and let $q \in V(u)$ and $r \in V(v)$ be the markers of the edge $uv$ in $T$.

1. If there is a $t \in V(u) - \{q\}$ that is mixed, then $r$ is mixed;
2. $r$ is empty if and only $V(u) - \{q\} \subseteq E(u)$;
3. $r$ is perfect if and only if $N[q] = P(u) \cup \{q\}$ and $V(u) - N[q] \subseteq E(u)$;

In the future, these remarks will be used without reference. Our interest in assigning states is to determine the existence of nodes of the types defined below:

**Definition 5.1.4.** Let $u$ be a node in a GLT. Then $u$ is fully-mixed if every $q \in V(u)$ (whose opposite is not a leaf) is mixed. A node $u$ is unmixed if every $q \in V(u)$ is either perfect or empty.

We will show that the changes to $ST(G)$ depend on the existence of an unmixed node. If such a node exists, then the changes will be local to that node. If such a node does not exist, then we will show that there must exist a fully-mixed node, and the changes in this case will be restricted to the GLT induced by all fully-mixed nodes. Below we explain how to identify unmixed and fully-mixed nodes.

### 5.1.2 Pruning: A Reduction

One can imagine that portions of $ST(G + (x, S))$ will be unchanged from $ST(G)$. This section is concerned with identifying them. We will show that if $q$ is a marker vertex that is either perfect or empty, then $T(q)$ will remain essentially unchanged from $ST(G)$ to $ST(G + (x, S))$. One way to see this is in terms of adjacencies. In a sense, if $q$ is empty or perfect, then $q$ “represents” $x$ in $T(q)$, meaning $x$ can safely be added to the rest of the split-tree without $T(q)$ needing to change. Another way of seeing this is in terms of splits.

Let $q \in V(u)$ be a perfect or empty marker vertex whose opposite, $r$, is not a leaf. Then by remark 3.3.12, $(L(q), L(r))$ is a split in $G$. Because $q$ is either perfect or empty, $(L(q), L(r) \cup \{x\})$
is a split in $G + (x, S)$. Intuitively, by applying lemma 3.3.18, we can ignore $T(q)$ when adding $x$, instead focusing on the rest of $ST(G)$. In the worst case, a single clique/star-join involving $u$ may afterwards be necessary.

Algorithm 5.1.2 recursively discards all nodes in trees $T(q)$ where $q$ is perfect or empty. What is left is either a single unmixed node, if one exists, or the split-tree’s fully-mixed nodes. The lemmas below combine to prove the correctness of algorithm 5.1.2.

Algorithm 5.1.2: Pruning(($T$, $F$), $S$)

**Input:** A GLT ($T$, $F$), and a subset $S$ of its leaves, $|S| > 1$.

**Output:** A set $U$ consisting of: a single unmixed node, if one exists; otherwise, the subset of $T$’s nodes that are fully-mixed with respect to $S$.

1. $U \leftarrow$ the nodes of $T(S)$;
2. set each node in $U$ to be active;
3. **foreach** active $u \in U$ with a unique neighbour in $U$ **do**
   1. set $u$ to be inactive;
   2. let $v$ be $u$’s unique neighbour in $U$;
   3. let $r$ be the marker of $uv$ in $v$;
   4. if $r$ is perfect **then** $U \leftarrow U \setminus \{u\}$;
4. **endfor**
5. return $U$;

**Lemma 5.1.5.** Let ($T$, $F$) be a GLT, and let $S$ be a subset of its leaves. If there exists an unmixed node with respect to $S$, then there is one in $T(S)$.

**Proof.** Let $u$ be an unmixed node and assume it is not a node in $T(S)$. Consider the shortest path between $u$ and a node in $T(S)$, and let $v \neq u$ be the other endpoint of this path. Let $u'$ be the neighbour of $v$ on this path (possibly $u = u'$). Observe that $u'$ is not a node in $T(S)$, by construction. Let $q \in V(v)$ and $r \in V(u')$ be the markers of the edge $vu'$. Since $u$ is unmixed, $r$ must be either perfect or empty. But as $v$ is in $T(S)$, we conclude that $r$ is perfect. But then every marker vertex in $V(v) \setminus \{q\}$ is either perfect or empty. Of course, $q$ must be empty since $u'$ is not in $T(S)$. Therefore $v$ is unmixed.

**Lemma 5.1.6.** Let ($T$, $F$) be a GLT, and let $S$ be a subset of its leaves. If there exists an unmixed node with respect to $S$, then algorithm 5.1.2 returns such an unmixed node.
Proof. We can assume that $T(S)$ contains an unmixed node, by lemma 5.1.5. Thus, we can assume that $U$ contains an unmixed node immediately before the execution of the loop. Consider the first iteration of the loop after which $U$ does not contain an unmixed node. Let $u, v$ and $r$ be as in the loop, and let $q \in V(u)$ be the other extremity of the edge $uv$. Then $u$ is unmixed, which means that $q$ is either perfect or empty. But as $v$ is in $T(S)$, we conclude that $q$ is perfect. But then every marker vertex in $V(v) - \{r\}$ is either perfect or empty. Of course, $r$ is perfect. Therefore $v$ is unmixed, a contradiction.

**Lemma 5.1.7.** If algorithm 5.1.2 returns a single node $v$, then $v$ is unmixed.

**Proof.** Let $u$ be a neighbour of $v$, and let $q \in V(u)$ and $r \in V(v)$ be the markers of the edge $uv$. If $u$ is not in $T(S)$, then $r$ must be empty. So assume that $u$ is in $T(S)$. Obviously, $u$ was removed from $U$ during the loop at some point. Since $v$ always remains in $U$, this can only happen because $v$ is the unique neighbour of $u$ and $r$ is perfect. Repeating this argument for all neighbours, we conclude that $v$ is unmixed.

**Lemma 5.1.8.** The set of nodes $U$ returned by algorithm 5.1.2 induces a subtree of $T(S)$.

**Proof.** This follows because $T(S)$ is a tree, and every node removed from $U$ during the loop has a unique neighbour in $U$.

**Lemma 5.1.9.** Let $(T, F)$ be a GLT, and let $S$ be a subset of its leaves. If there is no unmixed node with respect to $S$, then there exists at least two fully-mixed nodes. Furthermore, in this case, algorithm 5.1.2 returns the set of $T$’s fully-mixed nodes.

**Proof.** Let $u$ be a node in the set $U$ returned by the algorithm. Consider a longest path starting from $u$ in the tree induced by $U$ (see lemma 5.1.8), and let $v$ be the other endpoint of this path. By lemma 5.1.7, we know that $u \neq v$. Notice that $v$ must have a unique neighbour in $U$, by choice of the path; furthermore, this unique neighbour, call it $v'$ (possibly $v' = u$), must be $v$’s neighbour on the path.

Let $q \in V(u)$ and $r \in V(v)$ correspond to the edges incident to $u$ and $v$, respectively, on this path; and let $r' \in V(v')$ be the opposite of $r$. Then $r'$ cannot be empty, since $v \in U$; it cannot be perfect, since $v$ was not removed by the loop; hence, $r'$ is mixed. It follows that $q$ is mixed. We can extend this argument to prove that the markers of every edge in the tree induced by $U$ are mixed.

**Lemma 5.1.10.** Algorithm 5.1.2 is correct.
Figure 5.1: Forming $ST(G + (x, S))$ from $ST(G)$ by replacing an edge with a degenerate node, as required by algorithm 5.1.3. We use “P” for “perfect” and “E” for “empty”.

**Proof.** An immediate consequence of lemmas 5.1.6 and 5.1.9.

Roughly speaking, based on the intuition supplied earlier, algorithm 5.1.2 reduces the addition of $x$ to two cases: where $ST(G)$ contains an unmixed node, and where every node in $ST(G)$ is fully-mixed. The parts that follow below show how $x$ is added in the two cases.

### 5.1.3 Existence of an Unmixed Node

This section shows how to compute $ST(G + (x, S))$ when $ST(G)$ contains an unmixed node. In this case, $ST(G + (x, S))$ is formed according to algorithm 5.1.3. Understanding the algorithm requires the following definition:

**Definition 5.1.11.** The following sets are defined for each degenerate node $u$:

$$
P^*(u) = \{q \in V(u) \mid q \text{ perfect and not the centre of a star}\},
$$

$$
E^*(u) = \{q \in V(u) \mid q \text{ empty, or } q \text{ perfect and the centre of a star}\}.
$$

Algorithm 5.1.3 describes three possible outcomes for $ST(G + (x, S))$; these correspond to each branch of the algorithm’s first conditional. In the first two cases, a tree-edge is replaced by a new degenerate node of degree three, adjacent to $x$; this is demonstrated in figure 5.1. In the second of these cases a clique/star-split is first required. Notice that both cases may afterwards require a clique/star-join. This recalls the intuition provided in section 5.1.2. In the third case, on the other hand, $x$ is added to a prime node with the result being prime; hence, no clique/star-join is possible.

The next three lemmas prove that algorithm 5.1.3 is well-defined.

**Lemma 5.1.12.** Let $u$ be an unmixed degenerate node such that $|NE(u)| > 1$. If there is no $q \in V(u)$ such that $N[q] = P(u) \cup \{q\}$, then $|P^*(u)| > 1$. 

68
Algorithm 5.1.3: \texttt{Unmixed}(ST(G), S, u, x)

\textbf{Input}: The split-tree $ST(G) = (T,F)$, a subset $S$ of its leaves, $|S| > 1$, a node $u$ that is unmixed with respect to $S$, and a vertex $x \notin V(G)$.

\textbf{Output}: The split-tree $ST(G + (x, S))$.

if $\exists q \in V(u)$ such that $N[q] = P(u) \cup \{q\}$ then
  let $e$ be the tree edge corresponding to $q$;
else if $u$ is degenerate then
  update $(T,F)$ by node-splitting $u$ with respect to $(P^*(u), V(u) - P^*(u))$;
  let $e$ be the tree edge created by the node-split;
else
  update $(T,F)$ by adding a marker vertex to $u$, opposite $x$, and adjacent precisely to $P(u)$;
  return $(T,F)$;
endif

let $q$ and $r$ be the markers of $e$;
if $q$ and $r$ are both perfect then
  update $(T,F)$ by replacing $e$ with a clique node $v$ adjacent to $x$;
else
  update $(T,F)$ by replacing $e$ with a star node $v$ adjacent to $x$, its centre opposite the marker of $e$ that is perfect;
endif
update $(T,F)$ by performing a single clique/star-join involving $v$, if possible;
return $(T,F)$;
Proof. As \(|NE(u)| > 1\), there are distinct marker vertices \(q, q' \in V(u)\) that must either be perfect or mixed. But since \(u\) is unmixed, \(q\) and \(q'\) must actually be perfect. Therefore, if \(u\) is a clique, then \(|P^*(u)| > 1\). Furthermore, if \(u\) is a star and \(|P^*(u)| \leq 1\), then one of \(q\) and \(q'\) is its centre, say \(q'\). But then \(q\) is such that \(N[q] = P(u) \cup \{q\}\).

Lemma 5.1.13. Let \(u\) be an unmixed degenerate node in \(ST(G)\). If there is no \(q \in V(u)\) such that \(N[q] = P(u) \cup \{q\}\), then \((P^*(u), V(u) - P^*(u))\) is a split in \(G(u)\).

Proof. By lemma 5.1.12, \(|P^*(u)| > 1\). Since any non-trivial bipartition of a degenerate graph’s vertices forms a split, it suffices to show that \(|V(u) - P^*(u)| > 1\). So assume otherwise for contradiction.

First consider the case where \(u\) is a clique. If \(|V(u) - P^*(u)| \neq 1\), then there exists at least one \(q \in V(u)\) such that \(N[q] = P(u) \cup \{q\}\), a contradiction.

Now consider the case where \(u\) is a star with centre \(c\). By definition, \(c \in V(u) - P^*(u)\), so if \(|V(u) - P^*(u)| \neq 1\), then \(V(u) - P^*(u) = \{c\}\). But then \(c\) would be such that \(N[c] = P(u) \cup \{c\}\).

Lemma 5.1.14. Let \(e\) be the edge that results from clique/star-splitting an unmixed node \(u\) with respect to \((P^*(u), V(u) - P^*(u))\). Let \(q \in V(v)\) and \(r \in V(v')\) be e’s markers, and assume that \(v'\) is such that \(P^*(u) \subset V(v')\). Then \(q\) is perfect, and \(r\) is either perfect or empty, the former precisely when \(u\) is a star whose centre is perfect.

Proof. Direct from the definitions involved.

Finally, to prove the correctness of algorithm 5.1.3, we need the following results, the first of which is an obvious consequence of the definitions involved:

Remark 5.1.15. Let \(u\) be an unmixed degenerate node such that \(|NE(u)| > 1\). Then \(|P(u)| > 1\).

Lemma 5.1.16. Let \(G\) and \(G + x\) be two connected graphs such that \((A \cup \{x\}, B)\) is a split in \(G + x\), but \((A, B)\) is not a split in \(G\). Then either \(x\) has a twin or \(x\) is a pendant.

Proof. Let \(A'\) and \(B'\) be the frontiers of the split \((A \cup \{x\}, B)\). Since \((A, B)\) is not a split in \(G\), we know that \(|A \cup \{x\}| = 2\). If \(A' = \{x\}\), then \(G\) is disconnected, a contradiction. If \(A' = \{x, y\}\), then \(y\) is a twin of \(x\). If \(A' = \{y\}, y \neq x\), then \(N(x) = \{y\}\), since \(G + x\) is connected. Therefore \(x\) is a pendant.

Lemma 5.1.17. Algorithm 5.1.3 is correct.
Proof. Consider the GLT $(T, \mathcal{F})$ returned by the algorithm. It is an easy exercise to verify that its accessibility graph is $G + (x, S)$ (simply use lemma 5.1.14). Thus, it suffices to prove that it is reduced.

Assume first that either there is a $q \in V(u)$ such that $N[q] = P(u) \cup \{q\}$ or that $u$ is degenerate. In both cases it is clear that every label is either prime or degenerate. Moreover, since $ST(G)$ is reduced, the only possible clique-star/join must involve $v$, the newly inserted node. Therefore $(T, \mathcal{F})$ is reduced in this case.

Now consider the case where $u$ is prime and there is no $q \in V(u)$ such that $N[q] = P(u) \cup \{q\}$. Let $u_x$ be the result of adding a marker vertex $t$ to $u$, opposite $x$, and adjacent precisely to $P(u)$. If $u_x$ is not prime, then either $t$ has a twin or $t$ is a pendant, by lemma 5.1.16. If $t$ has a twin $q$, then $N[q] = P(u) \cup \{q\}$, a contradiction; and by remark 5.1.15, we know $d(t) > 1$. It follows that $u_x$ is prime. Since $u_x$ is prime, no clique/star-join is possible. Thus, $(T, \mathcal{F})$ is reduced in this case.

Algorithm 5.1.3 handles the case where $ST(G)$ contains an unmixed node. The changes when $ST(G)$ does not contain an unmixed node unfold in two parts, the first of which is explained next.

5.1.4 Nonexistence of an Unmixed Node

Cleaning: A Further Reduction

Recall from theorem 3.3.17, that degenerate labels in the split-tree encode many different splits in the underlying graph. The addition of a new vertex maintains some and destroys others. Those that are maintained must remain in the split-tree in the form of a degenerate label, while those that are destroyed cannot. Cleaning is the process of separating those splits that remain from those that do not. The ones that remain are summarized by $P^*(u)$ and $E^*(u)$. As we will see, the other splits (that are destroyed) will eventually be contracted (with other splits encoded by different parts of the split-tree) into a single prime node.

More precisely, consider the case where there is no unmixed node in $ST(G)$. Then $ST(G)$ contains a fully-mixed node, and if $U$ is the set of all such nodes, then $|U| > 1$, by lemma 5.1.9. The intuition from section 5.1.2 was that $x$ could be added to the GLT induced by $U$, with the other parts of $ST(G)$ remaining unchanged. The only caveat was that a single clique/star-join may afterwards be required to form $ST(G + (x, S))$. We show here that the portion of $ST(G)$ that remains unchanged can be expanded in the absence of an unmixed node.

Let $u$ be a degenerate node in $U$. Assume that $(P^*(u), V(u) - P^*(u))$ is a split in $G(u)$, and let $v$ and $v'$ be the nodes that result from node-splitting $u$ accordingly. The next lemma says that one of $v$ and $v'$ is fully-mixed and the other is not:
Lemma 5.1.18. Let $u$ be a fully-mixed degenerate node that is adjacent to at least one other node. Let $v$ and $v'$ be the nodes that result from clique/star-splitting $u$ with respect to $(P^*(u), V(u) - P^*(u))$. Choose $v$ to be such that $P^*(u) \subseteq V(v)$. Then $|V(v) - P^*(v)| \neq 1$ and $|P^*(v')| \neq 1$. Moreover, $v$ is fully-mixed but $v'$ is not.

Proof. Clearly, $V(v) = P^*(u) \cup \{q\}$, where $q$ is a new marker vertex created by the clique/star-split. But $P^*(u) \subseteq P^*(v)$, and therefore $|V(v) - P^*(v)| \neq 1$. It follows as well that $r$ (the opposite of $q$) is perfect. Hence, $v'$ is not fully-mixed.

Since $u$ is fully-mixed and adjacent to at least one other node, there is a $t \in V(u)$ that is mixed. Observe that $t \in V(v')$, and therefore $q$ is mixed. Then $v$ is fully-mixed, because $u$ was fully-mixed. Furthermore, if $v'$ contains a perfect marker vertex other than $r$, it can only be the centre of $u$’s star, and hence $|P^*(v')| \neq 1$. \[\square\]

Corollary 5.1.19. Let $u$ be a fully-mixed degenerate node that is adjacent to at least one other node. Let $v$ and $v'$ be the nodes that result from clique/star-splitting $u$ with respect to $(P^*(u), V(u) - P^*(u))$. Choose $v$ to be such that $V(v) = P^*(u) \subseteq V(v)$. Then $v'$ contains a perfect marker vertex whose opposite is not a leaf.

Proof. Immediate from the proof of lemma 5.1.18. \[\square\]

What follows are the $E^*(u)$ analogues of lemma 5.1.18 and corollary 5.1.19:

Lemma 5.1.20. Let $u$ be a fully-mixed degenerate node that is adjacent to at least one other node. Let $v$ and $v'$ be the nodes that result from clique/star-splitting $u$ with respect to $(E^*(u), V(u) - E^*(u))$. Assume $v$ is such that $E^*(u) \subseteq V(v)$. Then $|V(v) - E^*(v)| \neq 1$ and $|E^*(v')| \neq 1$. Moreover, $v$ is fully-mixed but $v'$ is not.

Proof. Clearly, $V(v) = E^*(u) \cup \{q\}$, where $q$ is a new marker vertex created by the clique/star-split. But $E^*(u) \subseteq E^*(v)$, and therefore $|V(v) - E^*(v)| \neq 1$. If $u$ was a clique, then clearly $r$ is empty. If $u$ is a star with centre $c$, and $c$ is perfect, then $r$ must be perfect; otherwise $r$ must be empty. Hence, $v'$ is not fully-mixed.

Since $u$ is fully-mixed and adjacent to at least one other node, there is a $t \in V(u)$ that is mixed. Observe that $t \in V(v')$, and therefore $q$ is mixed. Then $v$ is fully-mixed, since $u$ was fully-mixed. Furthermore, $v'$ cannot contain an empty marker vertex other than $r$, and therefore $|E^*(v')| \neq 1$. \[\square\]

Corollary 5.1.21. Let $u$ be a fully-mixed degenerate node that is adjacent to at least one other node. Let $v$ and $v'$ be the nodes that result from clique/star-splitting $u$ with respect to $(E^*(u), V(u) - E^*(u))$. Then $v'$ contains a perfect marker vertex whose opposite is not a leaf.
Figure 5.2: An example of cleaning a fully-mixed node $u$. The triangle represents a subtree of the GLT. The dashed lines delineate portions of the GLT that would be pruned as a result of newly created perfect/empty marker vertices. Labels have been assigned to certain marker vertices to indicate their state: “$P$” for “perfect”, “$E$” for “empty”, and “$M$” for “mixed”. The remaining labels represent leaves in the GLT.

Choose $v$ to be such that $V(v) = E^*(u) \subseteq V(v)$. Then $v'$ contains a perfect or empty marker vertex whose opposite is not a leaf.

**Proof.** Immediate from the proof of lemma 5.1.18.

Therefore, by corollaries 5.1.19 and 5.1.21, if either $(P^*(u), V(u) - P^*(u))$ or $(E^*(u), V(u) - E^*(u))$ is a split in $G(u)$, then we can clique/star-split accordingly and create new perfect and empty marker vertices. Recalling the intuition from section 5.1.2, this means we can expand the portion of $ST(G)$ that does not change. After the clique/star-split, the resulting perfect or empty marker vertex can be pruned, as was done in section 5.1.2. This is demonstrated in figure 5.2. It is this splitting process that we mean when we say **cleaning**, whose name is inspired by the following definition:

**Definition 5.1.22.** Let $u$ be a degenerate node. If neither $(P^*(u), V(u) - P^*(u))$ nor $(E^*(u), V(u) - E^*(u))$ is a split in $G(u)$, then $u$ is cleaned. A GLT is cleaned if every one of its degenerate nodes is cleaned.

The potential for cleaning is partially described by the following two lemmas:
Lemma 5.1.23. Let $u$ be a fully-mixed degenerate node with at least one neighbour $v$ that is fully-mixed. If $|P^*(u)| > 1$, then $(P^*(u), V(u) - P^*(u))$ is a split in $G(u)$.

Proof. Since any non-trivial bipartition of a degenerate graph’s vertices forms a split, it suffices to show that $|V(u) - P^*(u)| > 1$. So assume otherwise for contradiction.

Let $q \in V(u)$ and $r \in V(v)$ be the markers of the edge $uv$. Both $q$ and $r$ must be mixed since $u$ and $v$ are fully-mixed. Therefore $V(u) - P^*(u) = \{q\}$, and $P^*(u) = V(u) - \{q\}$.

Therefore $u$ cannot be a clique, for then $r$ would be perfect, a contradiction. So assume that $u$ is a star. Recall that the centre of a star is never in $P^*(u)$. Hence, $q$ must be $u$’s centre. But then $r$ would have to be perfect, a contradiction. \qed

Lemma 5.1.24. Let $u$ be a fully-mixed degenerate node with at least one neighbour $v$ that is fully-mixed. If $|E^*(u)| > 1$, then $(E^*(u), V(u) - E^*(u))$ is a split in $G(u)$.

Proof. Since any non-trivial bipartition of a degenerate graph’s vertices forms a split, it suffices to show that $|V(u) - E^*(u)| > 1$. So assume otherwise for contradiction.

Let $q \in V(u)$ and $r \in V(v)$ be the markers of the edge $uv$. Both $q$ and $r$ must be mixed since $u$ and $v$ are fully-mixed. Therefore $V(u) - E^*(u) = \{q\}$, and $E^*(u) = V(u) - \{q\}$.

Therefore $u$ cannot be a clique, for then $r$ would be empty, a contradiction. So assume that $u$ is a star. First assume that $q$ is $u$’s centre. Notice that $r$ would be empty in this case, a contradiction. So assume that $q$ is an extremity of $u$. Recall that if $c$ is $u$’s centre, then $c$ can be perfect or empty and still belong to $E^*(u)$. In the former case, $r$ would be perfect, and in the latter case $r$ would be empty, a contradiction either way. \qed

Roughly speaking, cleaning, along with algorithms 5.1.2 and 5.1.3, reduces the addition of $x$ to the case where $ST(G)$ is cleaned, does not contain an unmixed node, and all of its nodes are fully-mixed. We handle that case next before integrating all of these parts into a complete algorithm summarizing how $ST(G + (x, S))$ is formed from $ST(G)$.

Contraction

Assume that $ST(G)$ is cleaned, does not contain an unmixed node, and all of its nodes are fully-mixed. We will show that $G + (x, S)$ is prime in this case. Assume for contradiction that $(A \cup \{x\}, B)$ is a split in $G + (x, S)$. Recall our assumption that $|S| > 1$. Therefore, combining lemma 5.1.16 with the one below, we conclude that $(A, B)$ must be a split in $G$:

Lemma 5.1.25. Let $(T, \mathcal{F})$ be a GLT, and let $S$ be a subset of its leaves. If there is a leaf $y$ such that $N[y] = S \cup \{y\}$ (possibly $y \in S$), then $(T, \mathcal{F})$ contains an unmixed node.
Proof. Let \( q \in V(u) \) be the opposite of \( y \). Notice that \( q \) is either perfect or empty. Let \( t_1, \ldots, t_k \) be the neighbours of \( q \) in \( G(u) \). Observe that \( N(y) = A(t_1) \cup \cdots \cup A(t_k) \). But we know that \( N(y) = S - \{y\} \), hence, \( S - \{y\} = A(t_1) \cup \cdots \cup A(t_k) \). As the \( A(t_i) \)’s partition \( S - \{y\} \), this implies each \( t_i \) is perfect, by definition. On the other hand, if \( r \) is a non-neighbour of \( q \) in \( G(u) \), then \( N(y) \cap L(q) = \emptyset \). Thus, \( S - \{y\} \cap L(q) = \emptyset \). It follows that \( r \) is empty, by definition. Therefore every marker vertex of \( u \) is either perfect or empty, meaning \( u \) is unmixed.

It follows that \((A, B)\) is a split in \( G \). But then it would be encoded by \( ST(G) \) in one of two possible ways, corresponding to the two conditions in theorem 3.3.17: either the split corresponds to an edge in the split-tree, or it corresponds to a split in a degenerate node in the split-tree. Roughly speaking, the fact that every node in \( ST(G) \) is fully-mixed precludes the split from being encoded as in condition 1, and the fact that \( ST(G) \) is cleaned precludes the split from being encoded as in condition 2. Based on this, \( G + (x, S) \) must actually be prime. That is the intuition for algorithm 5.1.4, which describes how \( ST(G + (x, S)) \) is formed from \( ST(G) \) in this case. A formal proof of its correctness appears below, from which \( G + (x, S) \) being prime follows as a corollary.

Algorithm 5.1.4: Contraction\((ST(G), S, x)\)

\begin{verbatim}
Input: The cleaned split-tree \( ST(G) \) containing more than one node, all of its nodes being fully-mixed, none of them being unmixed; a subset \( S \) of its leaves, \(|S| > 1\); and a vertex \( x \notin V(G) \).

Output: The split-tree \( ST(G + (x, S)) \).

\((T, \mathcal{F}) \leftarrow ST(G)\);

\textbf{foreach} pair of adjacent nodes \( u \) and \( v \) in \((T, \mathcal{F})\) \textbf{do}
  \hspace{1em}update \((T, \mathcal{F})\) by performing the node-join of \( u \) and \( v \);
\textbf{endfch}

let \( u \) be the only node in \((T, \mathcal{F})\);

update \((T, \mathcal{F})\) by adding a marker vertex to \( u \), opposite \( x \), and adjacent precisely to \( P(u) \);

return \((T, \mathcal{F})\);
\end{verbatim}

Lemma 5.1.26. Algorithm 5.1.4 is correct.

Proof. Let \((T, \mathcal{F})\) be the GLT returned by the algorithm. It is an easy exercise to verify that its accessibility graph is \( G + (x, S) \). Thus, it suffices to prove that it is reduced. However, \((T, \mathcal{F})\) contains only one node whose label is isomorphic to \( G + (x, S) \). Hence, it suffices to prove that \( G + (x, S) \) is prime.
Assume for contradiction that \( G + (x, S) \) is not prime. Let \( (A \cup \{ x \}, B) \) be a split in \( G + (x, S) \) such that \( x \notin A, B \). If \( (A, B) \) is not a split in \( G \), then either \( x \) has a twin or \( x \) is a pendant, by lemma 5.1.16. With \( |S| > 1 \), we know that \( x \) is not a pendant. On the other hand, if \( x \) has a twin, then \( ST(G) \) contains an unmixed node, by lemma 5.1.25, a contradiction. Thus, \( (A, B) \) must be a split in \( G \). By theorem 3.3.17, there are two cases to consider:

Case 1: Assume that \( ST(G) \) contains a tree edge whose markers \( q \) and \( r \) are such that \( L(q) = A \) and \( L(r) = B \). Say \( q \in V(u) \) and \( r \in V(v) \), and observe that \( u, v \in U \). Since \( (A \cup \{ x \}, B) \) is a split, it follows that \( r \) is either perfect or empty. But then the nodes in \( ST(G) \) are not all fully-mixed, a contradiction.

Case 2: Assume that \( ST(G) \) permits a clique/star-split whose resulting tree edge has markers \( q \) and \( r \) such that \( A = L(q) \) and \( B = L(r) \). Let \( u \) be the node that was clique/star-split, and let \( v \) and \( v' \) be the nodes that result from the split. Say \( q \in V(v) \) and \( r \in V(v') \). Since \( (A \cup \{ x \}, B) \) is a split, it follows that \( r \) is either perfect or empty.

Suppose that \( v \) is a star. Consider the case where \( q \) is \( v \)'s centre. If \( r \) is perfect, then all of \( v \)'s extremities must be perfect, and therefore \( |P^*(v)| > 1 \). If \( r \) is empty, then all of \( v \)'s extremities must be empty, and therefore \( |E^*(v)| > 1 \). Each of \( v \)'s extremities is an extremity of \( u \). Therefore, either \( |P^*(u)| > 1 \) or \( |E^*(u)| > 1 \). Now, by lemma 5.1.9, since \( ST(G) \) contains no unmixed node, \( u \) must have a fully-mixed neighbour. So by lemmas 5.1.23 and 5.1.24, we contradict \( u \) being cleaned.

Now consider the case where \( q \) is an extremity of \( v \). If \( r \) is perfect, then \( v \)'s centre is perfect and all of its other extremities (except possibly \( q \)) are empty. Thus, \( |E^*(v)| > 1 \). If \( r \) is empty, then so is \( v \)'s centre, as well as every one of its extremities (except possibly \( q \)). Hence, \( |E^*(v)| > 1 \) once more. The desired contradiction now follows as above.

Suppose instead that \( v \) is a clique. Then if \( r \) is perfect, every marker vertex in \( V(v) \) (except possibly \( q \)) is perfect, meaning \( |P^*(v)| > 1 \). If \( r \) is empty, then every marker (except possibly \( q \)) is empty, meaning \( |E^*(v)| > 1 \). Of course, every marker vertex in \( V(v) \) \( - \{ q \} \) is a marker vertex of \( u \). Hence, either \( |P^*(u)| > 1 \) or \( |E^*(u)| > 1 \). The contradiction now follows as in the star case.

\[ \square \]

**Corollary 5.1.27.** Assume that \( ST(G) \) is cleaned, does not contain an unmixed node, and all of its nodes are fully-mixed. Then the node adjacent to \( x \) in \( ST(G + (x, S)) \) is prime.

**Proof.** Immediate from the proof of lemma 5.1.26. \[ \square \]

Recall the intuition from section 5.1.2. In particular, the fact that after adding \( x \) to the GLT induced by the unmixed/fully-mixed node(s), a single clique/star-join may afterwards be necessary involving \( x \)'s neighbour. We saw this arise in algorithm 5.1.3 for the unmixed case. However, the
Figure 5.3: An example of cleaning under the assumption that states are assigned with respect to the set $S = \{c, h, g, e, f, k\}$. Labels have been assigned to certain marker vertices to indicate their state: “$P$” for “perfect”, “$E$” for “empty”, and “$M$” for “mixed”. On the left, the delimited box represents the set of fully-mixed nodes. Notice that the degenerate node within the box is not cleaned. The effect of cleaning is demonstrated on the right, with the nodes in the delimited box now being cleaned and fully-mixed.

The fact that $x$ is adjacent to a prime node in this instance means it is unnecessary for the fully-mixed case. After the reductions of pruning and cleaning, these are the only two cases. We integrate these parts below to completely specify how $ST(G + (x, S))$ is formed from $ST(G)$.

### 5.1.5 Combining the Cases

After pruning, or rather, after algorithm 5.1.2, we are either left with a single unmixed node, or a set of fully-mixed nodes. In the former case, algorithm 5.1.3 describes how $ST(G)$ is updated to form $ST(G + (x, S))$. In the latter case, cleaning further reduces the problem, and on what remains we can apply algorithm 5.1.4 to update $ST(G)$. Of course, if $|S| = 1$, then we instead use algorithm 5.1.1. This process is summarized as algorithm 5.1.5; an example appears in figures 5.3 and 5.4.

Cleaning amounts to the two loops in algorithm 5.1.5; lemmas 5.1.23 and 5.1.24 imply that each is well-defined. The next lemma is needed to prove that a single GLT is induced by the set
Algorithm 5.1.5: AddVertex\(\left(ST(G), x, S\right)\)

**Input:** The split-tree \(ST(G)\), a non-empty subset \(S\) of its leaves, and a new vertex \(x \not\in V(G)\).

**Output:** The split-tree \(ST(G + (x, S))\).

if \(S = \{y\}\) then return Pendant\(\left(ST(G), y\right)\); // algorithm 5.1.1

\[U \leftarrow \text{Pruning}(ST(G), S); // \text{algorithm 5.1.2}\]

if \(U = \{u\}\) then return Unmixed\(\left(ST(G), S, u, x\right)\); // algorithm 5.1.3

\[(T, \mathcal{F}) \leftarrow ST(G); U' \leftarrow U;\]

**foreach** degenerate node \(u \in U'\) such that \(|P^*(u)| > 1\) do

update \((T, \mathcal{F})\) by node-splitting \(u\) with respect to \(\left(P^*(u), V(u) - P^*(u)\right)\);

let \(u'\) be the node resulting from the node-split such that \(V(u) - P^*(u) \subset V(u')\);

remove \(u\) from \(U'\) and replace it with \(u'\);

**endforeach**

**foreach** degenerate node \(u \in U'\) such that \(|E^*(u)| > 1\) do

update \((T, \mathcal{F})\) by node-splitting \(u\) with respect to \(\left(E^*(u), V(u) - E^*(u)\right)\);

let \(u'\) be the node resulting from the node-split such that \(V(u) - E^*(u) \subset V(u')\);

remove \(u\) from \(U'\) and replace it with \(u'\);

**endforeach**

let \((T', \mathcal{F}')\) be the GLT induced by the nodes in \(U'\);

let \(S'\) be the set of leaves in \(T'\) that are opposite perfect marker vertices;

update \((T, \mathcal{F})\) by replacing \((T', \mathcal{F}')\) with Contraction\((T', \mathcal{F}', S', x)\); // algorithm 5.1.4

**return** \((T, \mathcal{F})\);
Lemma 5.1.29. \textit{component are suitably met:} we only have to concentrate on proving that the input conditions of each node of figure 5.3 are contracted into a single node. On the right, a new vertex $x$ adjacent to \{c, h, g, e, f, k\} is added. Labels have been assigned to certain marker vertices with respect to this set of neighbours; we use “$P$” for “perfect”, “$E$” for “empty”, and “$M$” for “mixed”. 

$U'$ from algorithm 5.1.5:

\textbf{Lemma 5.1.28.} Assume that $ST(G)$ does not contain an unmixed node, and let $U$ be the set of its fully-mixed nodes. Expand $U$ by recursively clique/star-splitting its degenerate nodes according to $(P^*(u), V(u)−P^*(u))$ (if possible) and $(E^*(u), V(u)−E^*(u))$ (if possible) until no further node-split is possible. Let $U'$ be the subset of $U$ (after its expansion) consisting of $U$’s prime nodes and its non-fully-mixed degenerate nodes. Then $U'$ induces a subtree of $T(S)$.

\textit{Proof.} If $ST(G)$ does not contain an unmixed node, then it contains a fully-mixed node, by lemma 5.1.9. Furthermore, that lemma says that we can assume the set $U$ (prior to its expansion) is computed by algorithm 5.1.2. Thus, the set $U$ (prior to its expansion) induces a subtree of $T(S)$, by lemma 5.1.8. Since every one of its nodes is fully-mixed, every tree edge induced by $U$ at this point will have both its markers mixed. Of course, every marker vertex opposite a leaf is either perfect or empty. The result now follows from lemmas 5.1.18 and 5.1.20.

The correctness of algorithm 5.1.5 is straightforward, given the correctness already proven of its constituent parts. We only have to concentrate on proving that the input conditions of each component are suitably met:

\textbf{Lemma 5.1.29.} \textit{Algorithm 5.1.5 is correct.}
Proof. Assume that algorithm 5.1.2 returns the set $U = \{u\}$. By lemma 5.1.6, we know that $u$ is unmixed. Hence, the input conditions for algorithm 5.1.3 are clearly met. So by lemma 5.1.10, algorithm 5.1.5 is correct in this instance.

So assume that algorithm 5.1.2 returns a set $U$ containing more than one node. Then by lemma 5.1.9, $U$ is the set of $ST(G)$’s fully-mixed nodes. Hence, when $U'$ is initialized it becomes the set of $ST(G)$’s fully-mixed nodes in this case.

Consider $U'$ immediately after the two loops in algorithm 5.1.5 have completed. By combining lemmas 5.1.18 and 5.1.20 with the above observation, we conclude that the GLT $(T, F)$ input to algorithm 5.1.4 is both cleaned and fully-mixed. The fact that it is the split-tree for its underlying accessibility graph is obvious. Hence, the input conditions for algorithm 5.1.4 are met. So by lemma 5.1.26, algorithm 5.1.5 is correct in this instance as well.

The next result will be important later when proving the running-time of algorithm 5.1.5:

**Lemma 5.1.30.** There are $O(|T(S)|)$ node-joins performed by algorithm 5.1.4 during algorithm 5.1.5.

Proof. Each node in $(T', F')$ can be identified with a node in $U$, which induces a subtree in $T(S)$, by lemma 5.1.8. There is one node-join for every tree edge in $T'$.

Future proofs will also be facilitated by reframing algorithm 5.1.5 as an incremental characterization of the split-tree (and split-decomposition), as we do next.

### 5.1.6 Incremental Characterization

We are interested here in characterizing how nodes in $ST(G + (x, S))$ are formed. Algorithm 5.1.5 operates in place, directly updating portions of $ST(G)$ in its construction of $ST(G + (x, S))$. Many of the marker vertices and nodes in $ST(G)$ are not affected by these changes and persist into $ST(G + (x, S))$. The next two definitions formalize what is meant by this:

**Definition 5.1.31.** Suppose that $U$ is such that it consists of a single unmixed node from $ST(G)$ if one exists, otherwise it consists of $ST(G)$’s fully-mixed nodes, of which there is more than one. Let $q \in V(u)$ be a marker vertex in $ST(G)$. Then we say that $ST(G + (x, S))$ inherits $q$ from $ST(G)$ unless $u \in U$ and $q$ is mixed. Let $u$ be a node in $ST(G)$. Then we say that $ST(G + (x, S))$ inherits $u$ from $ST(G)$ if there exists a node $u'$ in $ST(G + (x, S))$ such that $V(u) = V(u')$.

**Definition 5.1.32.** Let $u$ be a degenerate node in $ST(G + (x, S))$ not adjacent to $x$. Then we say that $u$ is an offspring of $ST(G)$ if there exists a $q \in V(u)$ such that:

1. neither $q$ nor its opposite is inherited from $ST(G)$;
2. if $r \in V(u) - \{q\}$, then $r$ and its opposite are inherited from $ST(G)$.

Intuitively, a marker vertex is inherited if its corresponding edge is not contracted by algorithm 5.1.4; a node is inherited if it is not in the set of fully-mixed nodes; and a degenerate node is an offspring if it arose from cleaning. We can now precisely characterize each node in $ST(G + (x, S))$ according to the mode of its creation. The first of the two lemmas below addresses prime nodes, the second addresses degenerate nodes:

**Lemma 5.1.33.** Suppose that $U$ is such that it consists of a single unmixed node from $ST(G)$ if one exists, otherwise it consists of $ST(G)$’s fully-mixed nodes, of which there is more than one. Let $u_x$ be the node adjacent to $x$ in $ST(G + (x, S))$. Then:

1. all prime nodes in $ST(G)$ are inherited by $ST(G + (x, S))$, except possibly those in $U$;
2. if $u$ is a prime node in $ST(G + (x, S))$, then either $u$ is inherited from $ST(G)$ or $u = u_x$;
3. if $U = \{u\}$ and $u$ is prime but not inherited by $ST(G + (x, S))$, then $u_x$ is prime, and $u_x$ is formed from $u$ by adding $x$ as a neighbour and making $x$’s opposite precisely adjacent to $P(u)$;
4. if $|U| > 1$, then $u_x$ is prime and formed as follows:

   (a) expand the set of nodes $U$ by recursively node-splitting each degenerate node $u \in U$ according to $(P^*(u), V(u) - P^*(u))$ and $(E^*(u), V(u) - E^*(u))$ until no further node-join is possible;

   (b) let $U'$ be the subset of $U$ (after its expansion) consisting of $U$’s prime nodes and its non-fully-mixed degenerate nodes.

   (c) creating a node $u$ by recursively contracting all tree edges (via node-joins) induced by the set $U'$;

   (d) adding $x$ as a neighbour of $u$ and making $x$’s opposite precisely adjacent to $P(u)$.

**Proof.** Condition 1 is obvious. Condition 3 follows from lemmas 5.1.6 and 5.1.17, along with an examination of algorithm 5.1.3. Condition 4 follows from lemma 5.1.9, corollary 5.1.27, and an examination of algorithm 5.1.4. Condition 2 is a consequence of conditions 1, 3, and 4.

**Lemma 5.1.34.** Let $u$ be a degenerate node in $ST(G + (x, S))$. Then either:

1. $u$ is adjacent to $x$, and if $u$ is a star, then $x$’s opposite is one of its extremities;
2. *u* is an offspring of \( ST(G) \);

3. *u* is inherited from \( ST(G) \).

**Proof.** The split-tree \( ST(G + (x, S)) \) is returned either by algorithm 5.1.1, algorithm 5.1.3, or algorithm 5.1.4. A simple examination of the algorithms involved in the first two cases proves the lemma.

So consider the case where algorithm 5.1.4 returns \( ST(G + (x, S)) \). Let \( U \) be the set of vertices returned by algorithm 5.1.2, and let \( U' \) be the set of nodes after the two loops defining cleaning have completed.

Let \( D \) be the set of degenerate nodes not in \( U \), and let \( D' \) be the set of degenerate nodes not in \( U' \). Corollary 5.1.27 implies that all degenerate nodes in \( ST(G + (x, S)) \) are in \( D' \). Clearly \( D \subseteq D' \).

Obviously, those in \( D \) are inherited from \( ST(G) \).

Let \( t' \in V(v') - \{q\} \), and let \( t \in V(u) \) be its opposite. As \( t' \) is not mixed, it follows that it is inherited by \( ST(G + (x, S)) \) from \( ST(G) \). Furthermore, \( t' \) not being mixed means \( u \) is not in \( U \), and therefore \( t \) is inherited by \( ST(G + (x, S)) \) from \( ST(G) \).

Finally, observe that neither \( q \) nor \( r \) exist in \( ST(G) \), meaning neither are inherited from \( ST(G) \).

Algorithm 5.1.5 can naturally be extended to an algorithm that computes \( ST(G) \) when input \( G \). Nothing was assumed of \( x \) above aside from its neighbourhood \( S \) being non-empty. The ordering of vertices in such an algorithm need only respect this property. This is the case for LBFS orderings (remark 3.5.1). We will show that following an LBFS ordering allows for simplifications that permit an efficient implementation.

### 5.2 LBFS-Incremental Split Decomposition

The rest of this chapter is concerned with the problem of constructing \( ST(G) \) incrementally when vertices are added according to an LBFS ordering. We call this the *LBFS-incremental construction of \( ST(G) \).* To simplify its discussion, we adopt the following notation:
Notation 5.2.1. Let $\sigma = x_1, \ldots, x_n$ be an LBFS of the graph $G$. Then $G_i = G[\{x_1, \ldots, x_i\}]$.

Solving the problem is straightforward at this point. We merely need to extend algorithm 5.1.5 in the natural way, but add vertices according to an LBFS. This approach is summarized as algorithm 5.2.1.

**Algorithm 5.2.1: LBFSIncrementalSplitTree($G$)**

**Input**: A graph $G$.

**Output**: The split-tree $ST(G)$.

$\sigma \leftarrow x_1, \ldots, x_n$, an LBFS of $G$;

$ST(G_0) \leftarrow$ empty GLT;

foreach $i = 1$ up to $n$ do

$S_i \leftarrow N(x_i) \cap V(G_i)$;

$ST(G_i) \leftarrow AddVertex(ST(G_{i-1}, x_i, S_i))$;

endfor

return $ST(G_n)$;

An LBFS ordering simplifies the task of incrementally constructing $ST(G)$ in two ways: (a) it regulates what vertices can be twins of the new vertex added at each step; and (b) it regulates how stars are formed in the split-tree. These simplifications are necessary to guarantee the efficiency of algorithm 5.2.1. We explore them each in turn below.

### 5.2.1 Regulating Twins

The idea of regulating twins is important to algorithm 5.1.3. In particular, it simplifies the evaluation of its first conditional by restricting the potential marker vertices $q$ for which the test $N[q] = P(u) \cup \{q\}$ can succeed. Intuitively, such a $q$ is a type of twin to $x$. To see why, merely consider the GLT induced by the node $u$ in algorithm 5.1.3; in that GLT, $q$’s opposite must be a twin of $x$. The relationship between split decomposition, twins, and LBFS is summarized in lemma 5.1.16 and the following result:

**Lemma 5.2.2.** Let $G$ and $G + x$ be two connected graphs and let $\sigma$ be an LBFS of $G + x$ in which $x$ appears last. If $G$ is prime and $x$ has a twin $y$, then $y$ is either universal in $G$ or $y$ is the penultimate vertex in $\sigma$.

**Proof.** First consider the case where $|V(G + x)| \leq 3$. Then since $G + x$ is connected, it must either be a star or a clique. Either way, it is easy to verify the result.
So assume that \(|V(G + x)| > 3\). Observe that \(y\) is unique since \(G\) is prime. As \(x\) and \(y\) are twins, we must have \(x \in S(y)\). Let \(B\) be the set of vertices appearing before \(y\) in \(\sigma\).

We first show that \(B\) is non-empty. Assume otherwise, meaning \(y\) is the first vertex in \(\sigma\). If \(N(y) = \emptyset\), then \(y\) is universal in \(G\). So assume otherwise. Now, the vertices in \(N(y)\) follow \(y\) in \(\sigma\) and appear before those in \(\overline{N(y)}\). As \(x\) is the last vertex in \(\sigma\) and universal to \(N(y)\) by virtue of being \(y\)'s twin, there must be a join between \(N(y)\) and \(\overline{N(y)}\), by lemma 3.5.4.

But now, unless \(|N(y)| = 1\), \((N(y), N(y) \cup \{y\})\) is a split in \(G\), contradicting \(G\) being prime. When \(|N(y)| = 1\), if \(|\overline{N(y)}| = 1\), then \(G\) is a star on three vertices; otherwise \((N(y) \cup \{y\}, \overline{N(y)})\) is a split in \(G\). In either case, \(G\) is not prime, a contradiction. Thus \(B\) is not empty.

As \(x\) appears last in \(\sigma\), \(B\) and \(S(y)\) partition \(V(G + x)\); moreover, if \(S(y) = \{x, y\}\), then \(y\) is the penultimate vertex in \(\sigma\). So assume that \(|S(y)| > 2\). If \(|B| > 1\), then \((B, S(y) - \{x\})\) is a split in \(G\), contradicting \(G\) being prime. Thus, \(|B| = 1\), meaning \(B\) consists of a single vertex \(s\) universal to \(S(y)\), and thus \(s\) is universal in \(G + x\). Now the proof proceeds as in the \(B = \emptyset\) case. Note that \(s \in N(y)\) but that \(N(y) \cap S(y)\) could be empty. \(\square\)

To apply the previous lemma to the node \(u\) in algorithm 5.1.3, we need to define an LBFS ordering for \(G(u)\). We start by looking at an ordering of \(V(u)\) induced by an LBFS ordering of \(G\):

**Definition 5.2.3.** Let \(\sigma\) be an LBFS of \(G\), and let \(u\) be a node in \(ST(G)\). Define the ordering \(\sigma(u)\) over \(V(u)\) as follows: \(q \in V(u)\) appears before \(r \in V(u)\) if and only if \(\exists x \in A(q)\) such that \(x <_{\sigma} y\), for all \(y \in A(r)\).

The next two lemmas combine to prove that \(\sigma(u)\) is an LBFS of \(G(u)\). Given this, we will call \(\sigma(u)\), the induced LBFS of \(u\). The ensuing corollary is needed to translate lemma 5.2.2 from \(G\) to \(G(u)\):

**Lemma 5.2.4.** Let \(\sigma\) be an LBFS of \(G\), and let \(q\) be a marker vertex in \(ST(G)\). Choose the vertex in \(A(q)\) appearing earliest in \(\sigma\) and call it \(z\). Let \(G' = G[(V(G) - L(q)) \cup \{z\}]\). Then \(\sigma[V(G')]\) is an LBFS of \(G'\).

**Proof.** For simplicity, let \(\sigma' = \sigma[V(G')]\). Assume for contradiction that \(\sigma'\) is not an LBFS of \(G'\).

Then by lemma 3.5.4, there are three vertices \(a <_{\sigma'} b <_{\sigma'} c\) with \(ac \in E(G')\) and \(ab \notin E(G')\), but no vertex \(d' <_{\sigma'} a\) such that \(d'b \in E(G')\), \(d'c \notin E(G')\).

Of course, by the same lemma, and since \(\sigma\) is an LBFS of \(G\), we know that \(a <_{\sigma} b <_{\sigma} c\) with \(ac \in E(G)\), \(ab \notin E(G)\), and there is a vertex \(d <_{\sigma} a\) such that \(db \in E(G), dc \notin E(G)\). Without loss of generality, assume that amongst all such vertices \(d\) is the one appearing earliest in \(\sigma\). Observe that we must have \(d \in L(q)\).
Assume for contradiction that \( d \in A(q) \). Of course, \( d \neq z \); furthermore, \( a, b, c \) is in \( L(q) \). Thus, \( z \) and \( d \) share the same adjacencies with respect to \( a, b, \) and \( c \). But by choice of \( z \), we have \( z <_d d \), contradicting our choice of \( d \). Hence, \( d \in L(q) - A(q) \). We arrive at the desired contradiction by considering the remaining cases:

**Case 1:** Assume that \( z \notin \{b, c\} \). We have \( b \notin L(q) \) by definition of \( \sigma' \), contradicting \( db \in E(G) \).

**Case 2:** Assume that \( z \in \{b, c\} \). We have \( a \notin L(q) \) by definition of \( \sigma' \), hence \( da \notin E(G) \). So by lemma 3.5.4 again, there is a vertex \( d_1 <_d d \) such that \( d_1 a \in E(G) \) but \( d_1 b \notin E(G) \). Note that \( d_1 \notin L(q) \), by choice of \( z \) and the fact that \( a \notin L(q) \) by assumption. Therefore \( d_1 d \notin E(G) \).

The same argument above can now be repeated with \( d_1 \) playing the role of \( a, d \) playing the role of \( b, \) and \( a \) playing the role of \( c \). The result is an infinite sequence of vertices \( d_1 \notin L(q), d_2 \notin L(q), \ldots \), a contradiction.

**Lemma 5.2.5.** Let \( \sigma \) be an LBFS of \( G \), and let \( u \) be a node in \( ST(G) \). Then \( \sigma(u) \) is an LBFS of \( G(u) \).

**Proof.** For each \( q \in V(u) \), associate the leaf in \( A(q) \) appearing earliest in \( \sigma \). Let \( (T, \mathcal{F}) \) be the GLT induced by \( u \). Observe that without loss of generality, we can assume that in \( (T, \mathcal{F}) \), each \( q \in V(u) \) is opposite its associated leaf. The desired result now follows by inductive application of lemma 5.2.4.

**Corollary 5.2.6.** Let \( \sigma \) be an LBFS of \( G \), and let \( u \) be a node in \( ST(G) \). Let \( S = \{y \in A(q) \mid q \in V(u), z \notin A(q), z <_d y \} \). Then \( \sigma[S] \) is an LBFS of \( G[S] \).

**Proof.** Follows from lemma 5.2.5 and the definition of \( \sigma(u) \), along with the fact that \( G[S] \) is isomorphic to \( G(u) \).

It is now possible to translate lemma 5.2.2 from \( G \) to \( G(u) \):

**Lemma 5.2.7.** Let \( u \) be as in algorithm 5.1.3, and assume that \( u \) is prime. If \( \exists q \in V(u) \) such that \( N[q] = P(u) \cup \{q\} \), then \( q \) is either the last vertex in \( \sigma(u) \), or \( q \) is universal in \( G(u) \).

**Proof.** Assume that such a \( q \in V(u) \) exists. Let \( (T', \mathcal{F}') \) be the GLT induced by the node \( u \), and let \( G' = G(T', \mathcal{F}') \). Obviously, \( G' \) is isomorphic to \( G(u) \), and therefore \( G' \) is prime.

We can assume, without loss of generality, that for each \( t \in V(u) \), the leaf opposite \( t \) in \( (T', \mathcal{F}') \) is the leaf in \( A(t) \) appearing earliest in \( \sigma \). Let \( y \) be the leaf opposite \( q \) in \( (T', \mathcal{F}') \). Let \( S' = S \cap L(T') \), and consider the graph \( G' + (x, S') \). Notice that \( x \) and \( y \) are twins in \( G' + (x, S') \).

Now, \( \sigma[S'] \) is an LBFS of \( G' \), by corollary 5.2.6. Let \( \sigma' \) be the same as \( \sigma[S'] \) but with \( x \) added to the end. Clearly, since \( x \) is last in \( \sigma \), we can conclude that \( \sigma' \) is an LBFS of \( G' + (x, S') \). So by
lemma 5.2.2, \( y \) is either the penultimate vertex in \( \sigma' \) or \( y \) is universal in \( G' + (x, S') \). Hence, \( y \) is either the last vertex in \( \sigma[S'] \) or is universal in \( G' \). But of course, this means that \( q \) is either the last vertex in \( \sigma(u) \) or is universal in \( G(u) \).

If \( u \) is prime, then the previous lemma restricts the candidate \( q \)'s for which the test \( N[q] = P(u) \cup \{q\} \) in algorithm 5.1.3 can succeed. The lemma says only the following marker vertices can pass the test:

**Definition 5.2.8.** Let \( u \) be a prime node in \( ST(G) \) and let \( \sigma \) be an LBFS of \( G \). Then \( \text{last}(u) \) is the marker vertex in \( V(u) \) that appears last in \( \sigma(u) \), and \( \text{universal}(u) \) is any marker vertex that is universal in \( G(u) \).

Now, any two vertices that are universal are obviously twins, and therefore the existence of any such pair induces a split. Hence, a prime graph can have at most one universal vertex, meaning the previous definition is well-defined.

Of course, if \( u \) is degenerate, then it is a simple matter to determine if there exists a \( q \) that passes the test. The problem is when \( u \) is prime, in which case the test would have to be applied to every vertex in \( V(u) \). Lemma 5.3.6 significantly improves the efficiency of algorithm 5.1.3 by limiting the marker vertices to which the test must be applied.

### 5.2.2 Regulating Stars

Consider the slice \( S(y) \) in an operation of algorithm 3.5.1 producing the LBFS ordering \( \sigma \). Every vertex \( z \in S(y), z \neq y \), has the same neighbours as \( y \) amongst the vertices appearing before \( y \) in \( \sigma \), by definition. This fact regulates to some degree how nodes are formed during the LBFS-incremental construction of the split-tree. The next three lemmas explore the impact on the creation of stars.

The first two of these lemmas are fairly technical. Nevertheless, they are merely different perspectives on the following simple observation: once a star is created by the addition of a new vertex, that star’s extremities are expanded maximally. That is, if \( c \) is the centre of the star, and some later vertex is added to \( L(c) \), then the star will not subsequently be expanded by adding new extremities. The key is that this fact can be used to limit the number of node-joins in which stars participate during algorithm 5.1.4, as the third lemma below proves. This will be needed to prove the efficiency of algorithm 5.2.1. It will be helpful to review the definitions of *inherit* and *offspring* from section 5.1.6 to understand the lemmas.

**Lemma 5.2.9.** Consider the LBFS-incremental construction of \( ST(G) \) with respect to \( \sigma \). Assume that \( x_i \) is opposite an extremity of a star in \( ST(G_i) \). Let \( x_k \) be the vertex in \( S(x_i) \) appearing latest
in $\sigma$. Then in all $ST(G_j), i \leq j \leq k$, there exists a star node and a subset of its extremities, $t_1, \ldots, t_\ell$, such that $L(t_1) \cup \cdots \cup L(t_\ell) = \{x_i, \ldots, x_j\}$.

Proof. The proof is by induction. The base is $i$, which holds by assumption. So assume the lemma holds up to some $j, i \leq j < k$.

Now, the vertices of $S(x_i)$ appear consecutively in $\sigma$. Let $B$ be the set of vertices appearing before $S(x_i)$ in $\sigma$. By definition, stars must be comprised of at least three vertices, meaning $|B| > 1$, for otherwise $x_i$ could not be adjacent to a star in $ST(G_i)$.

Let $S = \{x_i, \ldots, x_{j+1}\}$. Let $u_j$ be the star node in $ST(G_j)$ guaranteed by our induction hypothesis. We can also assume by our induction hypothesis that there is a subset of $u_j$’s extremities, $t_1, \ldots, t_\ell$, such that $L(t_1) \cup \cdots \cup L(t_\ell) = S$. Therefore $(B, S - x_{j+1})$ is a split in $G_j$ except possibly that $|S - x_{j+1}| = 1$. But $x_{j+1}$ has the same neighbours in $B$ as do the other vertices in $S$, by virtue of them all being in $S(x)$. Therefore $(B, S)$ is a split in $G_{j+1}$.

Let $u_i$ be the neighbour of $x_i$ in $ST(G_i)$. Recall that $u_i$ is a star, and let $c_i$ be its centre. Choose some $y \in A(c_i)$. Observe that $x_i$ and $y$ must be adjacent. It follows that $x_i \in S$ and $y \in B$ are on the frontiers of the split $(B, S)$ in $G_{j+1}$.

We can now apply lemma 3.3.18 to conclude the induction step. Let $q_y$ be the marker vertex opposite $y$ in $ST(G[S \cup \{y\}])$, and let $q_x$ be the marker vertex opposite $x_i$ in $ST(G[B \cup \{x_i\}])$. Let $(T', F')$ be the GLT that results from grafting $q_y$ and $q_x$, and let $e$ be the edge having $q_x$ and $q_y$ as its markers. By lemma 3.3.18, either $(T', F') = ST(G_{j+1})$, or $ST(G_{j+1})$ is obtained by a single star/clique-join involving $e$’s endpoints. The induction step follows because $ST(G[B \cup \{x_i\}]) = ST(G_i)$, and $x_i$ is opposite an extremity of a star in $ST(G_i)$. □

**Lemma 5.2.10.** Consider the LBFS-incremental construction of $ST(G)$ with respect to $\sigma$. Let $c$ be the centre of a star in $ST(G_i)$, and let $t$ be its opposite. Assume there is some $x_j, j > i$, such that $x_j$ is opposite an extremity of the star $v$ in $ST(G_j)$. Then there is no edge in $ST(G_j)$ with markers $q$ and $r$ such that:

1. $r$ is an extremity of $v$, and is not opposite $x_j$;
2. $A(t) \subseteq A(r)$;
3. $A(c) \subseteq A(q)$.

Proof. Suppose for contradiction that such an edge exists. Let $s$ be a neighbour of $c$ such that $L(s)$ does not contain the first vertex in $\sigma$. Stars have at least two extremities by definition, and so at least one such $s$ must exist. Choose the vertex $x_k \in L(s)$ appearing earliest in $\sigma$. If $x_k \notin A(s)$,
then \( x_k \) can have no neighbours appearing before it in \( \sigma \), contradicting remark 3.5.1. Therefore \( x_k \in A(s) \).

Let \( B \) be the set of vertices in \( \sigma \) appearing before \( x_k \). Observe that \( N(x_k) \cap B \subseteq A(c) \). Since \( A(c) \subseteq A(q) \), we know that \( x_k \) and \( x_j \) share the same neighbours in \( B \). Hence, \( x_j \in S(x_k) \). Then by lemma 5.2.9, we know that in \( ST(G_j) \) there exists a star node \( u \) and a subset of its extremities \( t_1, \ldots, t_\ell \) such that \( L(t_1) \cup \cdots \cup L(t_\ell) = \{ x_k, \ldots, x_j \} \).

No matter if \( v = u \) or \( v \neq u \), it follows that \( L(r) \subseteq \{ x_k, \ldots, x_j \} \). Hence, \( A(r) \subseteq \{ x_k, \ldots, x_j \} \), and thus, \( A(t) \subseteq \{ x_k, \ldots, x_j \} \). But by our LBFS-incremental construction of \( ST(G_i) \), this implies that \( A(t) = \{ x_k \} \). The contradiction follows because \( |A(t)| > 1 \) since star nodes must have at least two extremities, by definition.

\[ \square \]

**Lemma 5.2.11.** Consider the LBFS-incremental construction of \( ST(G) \) according to algorithm 5.2.1. Let \( c \) be the centre of a star in \( ST(G_i) \). Assume that \( c \) is inherited by \( ST(G_j), j > i \). If \( r \) is \( c \)'s opposite in \( ST(G_j) \), then \( d(r) > 1 \).

**Proof.** Assume for contradiction that \( d(r) = 1 \). Without loss of generality, assume that \( j \) is the smallest index for which this occurs. Since \( ST(G_j) \) is reduced, it follows that if \( r \in V(v) \), then \( v \) is a star. As \( j \) is the smallest index for which this occurs, then \( v \) cannot have been inherited from \( ST(G_{j-1}) \), nor can it be an offspring of \( ST(G_{j-1}) \). Hence, \( x_j \) is adjacent to \( v \), and its opposite is one of \( v \)'s extremities.

Let \( t \) be the opposite of \( c \) in \( ST(G_i) \). Then clearly \( A(t) \subseteq A(r) \). Let \( A_{c_i} = A(c) \) in \( ST(G_i) \) and let \( A_{c_j} = A(c) \) in \( ST(G_j) \). Then clearly \( A_{c_i} \subseteq A_{c_j} \). The contradiction now follows from lemma 5.2.10, with \( c \) also playing the role of \( q \).

\[ \square \]

Roughly speaking, the previous lemma says that once a star node participates in a node-join, it cannot participate in a node-join with another star in the future. This is helpful because stars are edge-sparse. With so few edges, there is little to “charge” the work required by the join. The previous lemma says that the other node participating in the join will not be a star, thereby offering relatively more edges to charge.

We do not use the word “charge” by accident. Later in this chapter we will present a charging argument to bound the running-time of algorithm 5.2.1. The bound will be in terms of the implementation we present next.
5.3 An Implementation

This section presents a detailed implementation of algorithm 5.2.1. Later we will show that perfect and mixed marker vertices can be identified with a unique neighbour of the vertex added at each step. Therefore, if they are “touched” a constant number of times at each step, then the overall cost is linear in the size of the graph. This same identification cannot be done for empty marker vertices. Consequently, the overriding concern in our implementation is to avoid “touching” empty marker vertices. We show that algorithm 5.2.1 can be implemented touching only a constant number of empty marker vertices at each step.

Below we introduce the split-tree data-structure required for the implementation. Afterwards, we describe how each part of algorithm 5.1.5 is implemented using the data-structure, and also derive the cost of doing so. The section that follows combines these costs in an aggregate analysis to conclude the running-time of algorithm 5.2.1.

5.3.1 The Data-Structure

We will arbitrarily root the split-tree being constructed at the leaf that is the first vertex inserted. Rooting the split-tree simplifies its traversal. The following definition arises:

**Definition 5.3.1.** Let $u$ be a node in a rooted GLT. Then $u$’s root marker vertex is its marker vertex corresponding to the edge shared with $u$’s parent.

The data-structure is a collection of two types of data-objects: node data-objects and label data-objects; one each for each node (and leaf) in the split-tree. Additionally, some node data-objects will need to be organized as a disjoint-set in a union-find data-structure [33]. These three facets are each explained below; figure 5.5 shows how they interact to encode the split-tree (recall definition 5.2.8 for the meaning of last and universal).

**Label Data-Objects**

Prime nodes will have their labels stored as an adjacency-list. Degenerate nodes will only maintain a list of their marker vertices. Every root marker vertex will maintain a pointer to its opposite. A leaf will be treated as a node containing itself as a single marker vertex; hence, its label data-object consists of a single marker vertex maintaining a pointer to its opposite. Additionally, each marker vertex will maintain a variable for its state.
Figure 5.5: A parent and child in the split-tree, alongside their data-structure encoding. States are represented by “p” for “perfect”, “e” for “empty”, and “m” for “mixed”.

Node Data-Objects

The node data-object corresponding to the node (or leaf) \( u \) is a collection of pointers:

- a pointer to the root marker vertex in \( u \)’s corresponding label data-object;
- pointers to last\((u)\) and universal\((u)\), when \( u \) is prime;
- a pointer to its centre (in the corresponding label data-object), if \( u \) is a star;
- a pointer to the node data-object corresponding to \( u \)’s parent.

The last pointer requires further discussion. In our implementation, not every node will use its parent pointer; some of these pointers will be inactive. Precisely which ones are inactive depends on our use of union-find, which is explained below in detail.

Union-Find

Union-find is an abstract data-type representing a dynamically changing collection of disjoint-sets over a fixed universe of elements. It assumes that each of these sets has one of its members distinguished as the set-representative. It supports three operations:
- **initialize**(*x*): initializes the singleton set \{*x*\} whose set-representative is *x*;

- **find**(*x*): returns the set-representative of the set containing *x*;

- **union**(*S*₁, *S*₂): forms a set corresponding to \(S_1 \cup S_2\), returning its set-representative, which is chosen from amongst those for *S*₁ and *S*₂.

The fastest implementation of union-find can perform a collection of *k* union and find operations on a universe of *n* elements in time \(O(\alpha(n) \cdot k)\), where \(\alpha\) is the inverse of Ackermann’s function, a function so slowly growing as to be effectively constant (pg. 521-522 [33]). A single initialization operation takes constant time. We will make the simplifying assumption that the size of the universe is equal to the number of initialization operations.

We use a union-find data-structure to encode the children of prime nodes. That is, each disjoint-set corresponds to the children of some prime node; and each prime node’s set of children corresponds to a disjoint-set. We refer to these sets as follows:

**Definition 5.3.2.** The disjoint-sets in our data-structure will be called child-sets. If *u* is prime, the child-set corresponding to *u*’s children will be called *u*’s child-set.

Now we can describe how a node data-object’s parent pointer is used. Let *u* and *v* be adjacent nodes, with *u* the parent of *v*. If *u* is degenerate, then *v*’s parent pointer will correctly point to *u*. If *u* is prime, then *v*’s parent pointer will be **null** unless *v* is the set-representative of *u*’s child-set, in which case its parent pointer will correctly point to *u*.

We adopt this approach because otherwise node-joins would prove too costly to implement. If *u* and *v* undergo a node-join, then the children of *v* become children of *u*. We cannot afford to update the parent pointers of all of *v*’s children. That is why only set representatives maintain parent pointers (we ignore degenerate nodes for the time being). This problem of updating parent pointers is the algorithm’s bottleneck, of which further discussion is delayed until the conclusion of this thesis.

More practically, this manner of representing trees with union-find impacts their traversal. Traditionally, a bottom-up traversal would follow parent pointers up the tree. At each step, access to *u* would be obtained using *v*’s parent pointer. Access to *u* is now a two-step operation: parent(find(*v*)). Its cost will be addressed later when we undertake an aggregate analysis of algorithm 5.2.1.

Finally, we note that union-find does not support the deletion of any element from a set. We will have to be careful to avoid this because, logically at least, deletion from a set is required at certain points. In algorithm 5.1.3 for instance, a tree-edge is replaced by a new node. This has the
effect of removing a child from one node and replacing it with a new child. Another case occurs when a node is (logically) deleted during one of the node-joins in algorithm 5.1.4. we will see below that avoiding deletion in these cases is straightforward and requires no additional work.

5.3.2 Simplifications and Notation

Algorithm 5.2.1 begins by computing an LBFS $\sigma$ of its input graph $G$, and then repeatedly applies algorithm 5.1.5 to incrementally build the split-tree, adding vertices in the order of $\sigma$. It is well known that LBFS runs in linear-time, and any of its linear-time implementations will do for our purposes, for example [119, 85]. We instead focus on the implementation of algorithm 5.1.5. Our analysis will be stated in terms of its variables:

- the inputs $ST(G)$, $S$, and $x$;
- $U$, the set returned by algorithm 5.1.2, consisting of a single unmixed node, if one exists, otherwise consisting of all fully-mixed nodes;
- $(T, F)$, the GLT corresponding to $ST(G)$;
- $U'$ the set used in cleaning;
- $(T', F')$, the GLT, input to algorithm 5.1.4, induced by the set $U'$ after cleaning.

Given the incremental setting we have adopted, we can assume that $ST(G)$ has already been encoded by our data-structure. We can also assume that $x$ is an end-vertex for $G$, by remark 3.5.1.

We explained above that the cost of union-find was measured in aggregate, in terms of the total number of union, find, and initialization operations. To facilitate the analysis of algorithm 5.1.5, we adopt the following notation:

**Notation 5.3.3.** We will use $k_i$ to denote the cost of initialization, and use $k_f$ and $k_u$ to denote, respectively, the amortized cost of a find and union operation.

The above terms are placeholders. They serve to identify the cost of algorithm 5.1.5 that is due to union-find. We will use them to determine the total number of initialization, union, and find operations that are needed throughout algorithm 5.2.1. An immediate observation is the following:

**Remark 5.3.4.** Access from a child to its parent takes time $O(k_f)$ using our data-structure.

An implementation for each component of algorithm 5.1.5 is outlined below, along with its cost. We focus on the logical changes and describe how they must be implemented to guarantee the
algorithm’s efficiency. The technical details – like updating $|V(u)|$ for each node $u$, or updating individual pointers – and the justification of their cost are straightforward and left for the reader to verify. We refer the reader once more to figure 5.5 as a means of visualizing the implementation.

5.3.3 Updating last and universal

Lemma 5.1.33 describes how prime nodes are inherited by $ST(G + (x, S))$ from $ST(G)$, and how new prime nodes are created in $ST(G + (x, S))$. We use the result to show how last and universal are calculated for prime nodes in $ST(G + (x, S))$.

Let $u$ be a prime node in $ST(G + (x, S))$ inherited from $ST(G)$. Observe that $x$ cannot be adjacent to $u$, by lemma 5.1.33. Now, for any marker vertex $t \in V(u)$, the vertex in $A(t)$ appearing earliest in $\sigma$ does not change from $ST(G)$ to $ST(G + (x, S))$, since $x$ appears last in $\sigma$. As a result, last($u$) is the same in $ST(G + (x, S))$ as it was in $ST(G)$. Furthermore, since $G(u)$ remains unchanged between $ST(G)$ and $ST(G + (x, S))$, so does universal($u$). It follows that last($u$) and universal($u$) do not need to be updated for prime nodes in $ST(G + (x, S))$ that were inherited from $ST(G)$.

Consider instead a prime node in $ST(G + (x, S))$ not inherited from $ST(G)$. Such a prime node must be adjacent to $x$, by lemma 5.1.33. Let $u_x$ be this prime node. Since $x$ is last in $\sigma$, $x$’s opposite is last in $\sigma(u_x)$. Therefore last($u_x$) is equal to $x$’s opposite.

Now consider universal($u_x$). There are two cases, corresponding to conditions 3 and 4 of lemma 5.1.33. For condition 3, $u_x$ is formed by adding $x$ as a neighbour to a prime node $u$ in $ST(G)$. In this case, universal($u_x$) must either be universal($u$) or it must be $x$’s opposite. It is clearly a constant time operation to distinguish between these possibilities.

Assume now that condition 4 of lemma 5.1.33 describes the creation of $u_x$. Let $U'$ and $u$ be as defined in condition 4. Let $u'$ and $v'$ be adjacent nodes participating in one of the node-joins creating $u$, and let $v$ be the result. Given universal($u'$) and universal($v'$) it is clearly a constant time operation to determine universal($v$). Thus, we can assume that universal($u$) has been calculated. Either universal($u_x$) is universal($u$) or it is $x$’s opposite. Once more, it is clearly a constant time operation to distinguish between these possibilities.

Now, by lemma 5.1.30, there are $O(|T(S)|)$ node-joins performed by algorithm 5.1.4 during algorithm 5.1.5. The cost, therefore, to calculate universal($u_x$) is $O(|T(S)|)$. Combining this with our observations above, we conclude the following:

**Lemma 5.3.5.** The cost to update the values last and universal for all prime nodes is $O(|T(S)|)$. 
5.3.4 Pruning

Here we analyze the running-time of algorithm 5.1.2. The algorithm can be divided into two parts: (a) computing $T(S)$; and (b) the loop that follows. We first look at how each part is implemented and then consider the cost of this implementation.

Implementation

To compute $T(S)$, we use the algorithm from [77], which is repeated here as algorithm 5.3.1; our data-structure clearly supports its implementation.

Algorithm 5.3.1: [77] Computing the Subtree Spanning a Set of Leaves

| Input: A tree $T$ rooted at a leaf and a subset $S$ of its leaves, $|S| > 1$. |
| Output: The subtree $T(S)$ |

mark each leaf of $S$ as active;

while [the root is not visited and there are at least two active vertices] OR [the root is visited and there is at least one active vertex] do

foreach active node $u$ do

$u$ is no longer active: it becomes visited;

if $u$ is not the root and its parent is not visited then $u$’s parent is marked active;
endfch

endw

Let $T'$ be the subtree of $T$ induced by the active nodes;

if the root of $T'$ has a unique active child but does not belong to $S$ then

remove the path in $T'$ from the root to the closest node with at least two active children;

return $T'$;

Concerning the loop in algorithm 5.1.2, recall that just prior to its execution, $U$ is the set of nodes in $T(S)$. Therefore, a bottom-up scan of $T(S)$ allows us to compute, for each node in $U$, the number of its neighbours in $U$. This allows us to initialize the subset of nodes with exactly one neighbour in $U$. We will maintain this set during the loop: as each node is removed from $U$, its unique neighbour is checked to see if it now has a unique neighbour in $U$.

Let $u, v$, and $r$ be as in the loop, and let $q$ be $r$’s opposite. To determine if $r$ is perfect, we use the test: $r$ perfect if and only if $N[q] = P(u) \cup \{q\}$. This test alone is sufficient because of the following result:
Lemma 5.3.6. Let \( u, v \) and \( r \) be as in the loop in algorithm 5.1.2, and let \( q \in V(u) \) be \( r \)'s opposite. Then \( V(u) - \{q\} \subseteq P(u) \cup E(u) \).

Proof. The opposite of any leaf \( y \) adjacent to \( u \) must either be perfect or empty. So let \( u' \) be a node adjacent to \( u \) but not in \( U \), and let \( t \in V(u) \) be \( u \)'s marker of such an edge \( uu' \). If \( u' \) is not in \( T(S) \), then \( t \) must be empty. Otherwise \( u' \) is in \( T(S) \) but was removed during a previous iteration of the loop, in which case \( t \) must be perfect. \( \square \)

Nevertheless, in order to apply the test, we need to identify those marker vertices that are perfect. Initially, we scan the leaves in \( T(S) \) and mark their opposites as perfect. The remaining perfect marker vertices are identified during the loop, using the test. We can therefore assume the following:

Remark 5.3.7. Let \( u \) be a node returned by algorithm 5.1.2 as part of the set \( U \). Then we can assume that \( P(u) \) has been calculated.

It now becomes a simple matter to compute \( M(u) \) as well: the empty marker vertices are those whose corresponding edge was not in \( T(S) \). Of course, \( M(u) = V(u) - (E(u) \cup P(u)) \). All marker vertices whose corresponding edge is in \( T(S) \) can initially be marked as mixed. This can be done with a single bottom-up scan of \( T(S) \). Those marker vertices that are actually perfect can have their state updated during the loop, as just explained. Therefore:

Remark 5.3.8. Let \( u \) be a node returned by algorithm 5.1.2 as part of the set \( U \). Then we can assume that \( M(u) \) and \( NE(u) \) have been calculated.

The last two remarks are critical going forward. Most of the rest of the implementation relies on knowing the elements in the set \( P(u) \) or those in the set \( NE(u) \). In what follows, we mainly take their prior calculation for granted, assuming they are known \textit{a priori}.

Cost

As proved in [77], algorithm 5.3.1 can be implemented in time \( O(|T(S)|) \), given constant time access from a child to its parent. Given the same, the set of nodes with unique neighbours in \( U \) can clearly be maintained in time \( O(|T(S)|) \). And obviously, the bottom-up scan to label marker vertices in \( T(S) \) as mixed is \( O(|T(S)|) \) under the same assumption. Therefore, in all three cases, the number of times a parent must be accessed from its child is \( O(|T(S)|) \). In our data-structure, access from a child to its parent takes time \( O(k_f) \). Thus, the cost for the implementation of these three aspects is \( O(|T(S)| + k_f \cdot |T(S)|) \). Therefore:

\[ 95 \]
Now consider the cost of identifying the perfect marker vertices of nodes in $U$. The first scan over the leaves in $T(S)$ takes time $O(|T(S)|)$, while the test for perfection of $r$ clearly takes time $O(|P(u)|)$. The key observation is that this test is performed at most once for each node in $T(S)$: only when that node has a unique neighbour in $U$. The total time to perform these tests is therefore $O(p)$, where $p$ is the total number of perfect marker vertices in $T(S)$. But observe the following:

**Remark 5.3.9.** Every perfect or mixed marker vertex in $T(S)$ must have its corresponding edge in $T(S)$.

Therefore $p$ is $O(|T(S)|)$. Combining this conclusion with the one above, we get:

**Lemma 5.3.10.** Algorithm 5.1.2 takes time $O(|T(S)| + k_f \cdot |T(S)|)$.

Finally, before moving on, we make the following important remark:

**Remark 5.3.11.** Let $u$ be a node returned by algorithm 5.1.2 as part of the set $U$. Then we can assume that $u$’s parent can be accessed from $u$ in constant time.

The point is that the parent of every node in $U$ will have been identified during algorithm 5.1.2. These can be remembered in subsequent steps. This fact simplifies the analysis presented in the sections below, and will be implicitly assumed throughout.

### 5.3.5 Existence of an Unmixed Node

The analysis of algorithm 5.1.3 first considers how its different outcomes are determined, and then turns to the implementation of these outcomes.

**Identifying the Outcomes**

Let $u$ be as in the input to algorithm 5.1.3. Assume first that there is a $q \in V(u)$ such that $N[q] = P(u) \cup \{q\}$. Then $q$’s opposite is clearly perfect. Of course, every marker vertex in $P(u)$ and $NE(u)$ has already been determined.

If no such $q$ exists, but $u$ is degenerate, then we can apply lemma 5.1.14 to determine the states of the edge $e$ that results from the split $(P^*(u), V(u) - P^*(u))$. Doing so is clearly a constant time operation.

It follows from the above two observations that the states of the edge $e$ can be determined in constant time, and so the second conditional can be evaluated in constant time.

Now consider the first conditional. We can test if $u$ is degenerate in constant time given our data-structure. If $u$ is degenerate, then it is a simple matter to test in constant time if $\exists q \in V(u)$...
such that \( N[q] = P(u) \cup \{q\} \), given the prior calculation of \( P(u) \) and \( NE(u) \). When \( u \) is prime, we take advantage of \( x \) being an end-vertex, and apply lemma 5.3.6. The lemma says that there are at most two candidate \( q \)'s that need to be tested. Both of these can be accessed in constant time given our data-structure. Performing the test for each such \( q \) clearly takes time \( O(|P(u)|) \), which is \( O(|T(S)|) \), by remark 5.3.9. The next remark follows from this and our observations above:

**Remark 5.3.12.** It takes \( O(|T(S)|) \) time to determine what changes are required of \( ST(G) \) in algorithm 5.1.3.

**Replacing an Edge**

One of the outcomes of algorithm 5.1.3 is that a single edge in \( ST(G) \) is replaced by a degenerate node of degree three adjacent to \( x \), followed possibly by a single clique/star-join involving this new node. Here we describe the implementation and cost of this step.

The new degenerate node \( u \) has degree three and is adjacent to \( x \). So clearly its label data-object can be formed and integrated into the existing data structure in constant time. For its node data-object, we must be careful to respect our use of union-find.

Let \( v \) and \( v' \) be the endpoints of the edge \( e \) that is replaced, and assume that \( v \) is the parent of \( v' \). Logically, \( v' \) must be removed as a child of \( v \) and made a child of \( u \), and \( u \) must be made a child of \( v \). If \( v \) is degenerate, then this can be done in the obvious way in constant time. The problem exists when \( v \) is prime, because union-find does not support deletion. The trick is to reuse \( v' \)'s node data-object for \( u \), updating it as needed, and create a new node data-object for \( v' \). This can clearly be performed in constant time.

Finally, consider the possible star/clique-join involving \( u \). In fact, this clique/star-join is redundant. Notice that the result of the clique/star-join can also be obtained by directly adding \( x \) as a neighbour of \( v \) or \( v' \), without first replacing \( e \) with \( v \). Once again, this can clearly be performed in constant time.

**Attaching to an Existing Node**

Aside from replacing an edge, the other possible outcome of algorithm 5.1.3 is to add \( x \) as a neighbour of an existing node \( u \). Observe that in our rooted tree scenario, \( x \) is always made a child of \( u \). Now, if \( u \) is degenerate, this can be done in the obvious way in constant time.

So assume that \( u \) is prime. Then there are two steps: (a) \( x \)'s opposite must be added to \( u \)'s label data object; and (b) \( x \) must be integrated into \( u \)'s existing child-set. The former can be done
in the obvious way in time \(O(|P(u)|)\), which is \(O(|T(S)|)\), by remark 5.3.9. The latter requires initialize\((x)\), and then a union between \(u\)’s child-set and \(\{x\}\).

Combining the observations here with those in section 5.3.5, as well as remark 5.3.12:

**Lemma 5.3.13.** Algorithm 5.1.3 takes time \(O(|T(S)| + k_i + k_u)\).

### 5.3.6 Cleaning

Cleaning amounts to the two loops in algorithm 5.1.5. The first loop concerns splits of the form \((P^*(u), V(u) - P^*(u))\), and the second concerns splits of the form \((E^*(u), V(u) - E^*(u))\). Hence, we refer to the first loop as **perfect cleaning**, and refer to the second loop as **empty cleaning**. Below we concentrate on their implementation and then examine the resulting cost.

**Implementation**

For both perfect cleaning and empty cleaning, the implementation we choose starts by scanning the set \(U'\) to identify its degenerate nodes. For perfect cleaning, the test \(|P^*(u)| > 1\) is then performed. The test for empty cleaning would normally be \(|E^*(u)| > 1\). However, as this would require “touching” empty marker vertices to calculate \(|E^*(u)|\), we choose to implement an equivalent, but different test.

Notice that if \(u\) is a clique, then \(E^*(u) = V(u) - NE(u)\). In this case, we perform the test \(|V(u) - NE(u)| > 1\). So assume that \(u\) is a star with centre \(c\). First, we check \(c\) to learn its state. If perfect or empty, then \(E^*(u) = V(u) - (NE(u) - \{c\})\), otherwise \(E^*(u) = V(u) - NE(u)\). The corresponding tests are \(|V(u) - (NE(u) - \{c\})| > 1\) and \(|V(u) - NE(u)| > 1\), respectively.

To efficiently implement the clique/star-splits performed during cleaning we need to reuse existing data-objects. Let \(u\) be a node that is clique/star-split during cleaning, and let \(u'\) and \(u''\) be the nodes that result. Notice that the label data-objects for \(u'\) and \(u''\) can be created with minimal effort from that for \(u\). The idea is to remove elements from \(u\) to create \(u''\); what is leftover of \(u\) can be repurposed for \(u'\). An example is provided in figure 5.6.

We have to be careful of changes required in the tree, because of our use of union-find. As a result of the node-split, \(u\)’s parent may change. If \(u\)’s parent was previously prime, this would ostensibly involve deleting a node from a disjoint-set, something that is not allowed. However, we described a way around this in section 5.3.5: merely reuse \(u\)’s node data-object for the new parent.
Figure 5.6: Example implementation of a star-split during perfect cleaning. We use “P” to mark the marker vertices that are perfect. On the right, the labelled marker vertices are those that have been reused, while the unlabelled ones are newly created.

Cost

We have already calculated $P(u)$ and $NE(u)$ and can therefore assume that $|P(u)|$ and $|NE(u)|$ have also been calculated. $|V(u)|$ is encoded by the data structure, as is access to the centre of each star. Thus, the tests required by each loop can be implemented in constant time.

For each clique/star-split during perfect cleaning, the cost is clearly $O(|P^*(u)|)$, which is $O(|P(u)|)$, which in turn is $O(|NE(u)|)$. The cost for a clique/star-split during empty cleaning is analogously $O(|V(u) - E^*(u)|)$, which is $O(|NE(u)|)$.

Now, the set $U'$ is not constant during cleaning. However, every new node added to $U'$ replaces a degenerate node that has undergone a split. Therefore, at most $2 \cdot |U|$ nodes are ever processed during cleaning (recall that $U' = U$ before cleaning), and each of these is only ever subject to a single clique/star-split.

Combining the observations above, we conclude that the cost of cleaning is $O(|U| + k)$, where $k$ is the total number of marker vertices in $U$ that are either perfect or mixed. But recall that $U$ induces a subtree of $T(S)$, by lemma 5.1.8. Of course, if $u$ is a node in $T(S)$, and $q \in V(u)$ is perfect or mixed, then its corresponding edge is also in $T(S)$, by remark 5.3.9. Hence, $k$ is $O(|T(S)|)$. Therefore:

**Lemma 5.3.14.** Cleaning takes time $O(|T(S)|)$.
5.3.7 Contraction

Consider the node $u$ to which $x$ is added as a neighbour in algorithm 5.1.4. Let $u_x$ be $u$ after $x$ has been added as a neighbour, and $x$’s opposite has been added to $G(u)$; in other words, let $u_x$ be the node adjacent to $x$ in $ST(G + (x, S))$. Then by corollary 5.1.27, $u_x$ is prime. In terms of our data-structure, this means that $u_x$’s children must be encoded as a child-set, and its label must be represented by an adjacency list.

We will implement these two requirements in two stages. In the first, a child-set and adjacency list representation is created for $u$. The second consists of adding $x$ as a neighbour of $u$. The latter can be handled as in section 5.3.5. The implementation of the former will be divided into its logical parts: what is required for the child-set – the node data-objects – and what is required for the adjacency list – the label data-objects. Each is addressed in turn below.

Creating the Child-Set

First, we create a child-set for every degenerate node $u$ in $T'$. If $u$ has $d$ non-root marker vertices, this requires $d$ initialize operations, and $d$ union operations. Afterwards, just prior to the execution of the loop in algorithm 5.1.4, we can assume that every node in $T'$ has a child-set.

Let $u$ and $v$ be as in the loop in algorithm 5.1.4, and assume that $u$ is the parent of $v$. Logically, $u$ and $v$ are replaced by the result of the node-join, call it $u'$. We will instead repurpose the node data-object for $u$ as the node data-object for $u'$, and keep the node-data object for $v$. To effect this, we perform a single union operation between $u$’s child-set and $v$’s child-set. The set-representative of $v$’s child-set can no longer point to $v$, and may now need to point to $u$ (if it is chosen as the set-representative of the child-set that results from the union). Updating this pointer is clearly a constant time operation.

We cannot delete $v$’s node data-object because union-find does not support deletion. However, we lose nothing by keeping it. As $v$ no longer logically exists, its node data-object’s pointer to its label data-object can be removed. We have already removed the only parent pointer that could point to $v$ (the set-representative of $v$’s former child-set). In this way, $v$’s node data-object ceases to represent a node in the tree. It only persists for the functioning of union-find.

Combining the above observations with lemma 5.1.30:

**Remark 5.3.15.** Let $u$ be the node that results from the loop in algorithm 5.1.4 (after being called by algorithm 5.1.5). Then the child-set for $u$ can be created in time $O(k_u \cdot |T(S)| + d_v \cdot (k_i + k_u))$, where $d_v$ is the total number of non-root marker vertices of degenerate nodes in $T'$.$^2$

---

$^2$d: “$d$” for “degenerate” and “v” for “vertices”.
Creating the Adjacency-List

Once again, we begin by initializing each degenerate node \( u' \in T' \). Here, we must create the adjacency list representation for \( u' \). Doing so in the obvious way costs \( O(d') \), where \( d' \) is the number of label edges in \( G(u') \).

Next, we process each node-join in turn, just as we did to create the child-set. We define three types of node-joins:

**Definition 5.3.16.** Let \( u \) and \( v \) be as in the loop in algorithm 5.1.4, and assume that \( u \) is the parent of \( v \). Let \( q \in V(u) \) and \( r \in V(v) \) be the markers of the edge \( uv \). If \( d(r) = 1 \), then the node-join between \( u \) and \( v \) is a child-extremity node-join. If \( d(q) = 1 \) and \( d(r) > 1 \), then the node-join between \( u \) and \( v \) is a parent-extremity node-join. In all other cases, the node-join between \( u \) and \( v \) is a regular node-join.

It will help the implementation to perform the node-joins in the order: child-extremity node-joins, parent-extremity node-joins, regular node-joins. Doing so guarantees the following:

**Lemma 5.3.17.** Let \( u \) and \( v \) be as in definition 5.3.16. Then at the time a child-extremity node-join is performed, \( v \) is a star; and at the time a parent-extremity node-join is performed, \( u \) is a star.

**Proof.** This is a simple inductive argument based on the fact that \( ST(G) \) is reduced. \( \square \)

Hence the names given to non-regular node-joins. A single scan through \( T' \) suffices to categorize and prioritize the node-joins. Of course, each node in \( T' \) can be identified with a node in \( U \), and so the total number of nodes in \( T' \) is \( O(|T(S)|) \), by lemma 5.1.8. So the cost of this prioritization is \( O(|T(S)|) \).

Let \( u, v, \) and \( u' \) be as in the loop in algorithm 5.1.4, and assume that \( u \) is the parent of \( v \). Let \( q \in V(u) \) and \( r \in V(v) \) be the markers of the edge \( uv \). If the node-join between \( u \) and \( v \) is a parent-extremity node-join or a regular node-join, then the label data-object for \( u' \) will be created from those for \( u \) and \( v \) in the obvious way. The cost of the entire operation is clearly \( O(d(q) \cdot d(r)) \), due primarily to the following:

**Remark 5.3.18.** There are \( d(q) \cdot d(r) \) new label edges created as a result of parent-extremity node-joins and regular node-joins.

To guarantee the efficiency of our algorithm, we will need to implement child-extremity node-joins using the following procedure:
Figure 5.7: Example implementation of a child-extremity node-join during contraction. Labelled marker vertices are those that are reused. Notice the change in $T(d)$ from left to right.

- let $c$ be the centre of $v$ and $r$ its root marker vertex;
- remove each extremity $t \in V(v) - \{r\}$ from $v$ (taking $T(t)$ with it);
- add each of the removed extremities $t$ (taking $T(t)$ with it) to $u$ by making it adjacent to $q$;
- replace $T(q)$ with $T(c)$;
- delete $v$.

The correctness of the above procedure is straightforward; see figure 5.7 for an example. In effect, we are reusing $q$ to play the role of $c$. Doing so means fewer new label edges are created. Specifically:

**Remark 5.3.19.** There are $d(c) - 1$ new label edges created as a result of a child-extremity node-join.

As a result, the cost of the above approach is clearly $O(d(c))$.

We can now conclude the cost of algorithm 5.1.4. This requires combining remark 5.3.15 with the observations here and in section 5.3.5:

**Lemma 5.3.20.** Let $d_e$ be the total number of label-edges of degenerate nodes in $T'$; let $d_v$ be the total number of non-root marker vertices of degenerate nodes in $T'$; and let $e_n$ be the total number of new label-edges created by the node-joins performed during algorithm 5.1.4 (after being called by algorithm 5.1.5). Then algorithm 5.1.4 takes time $O(k_u \cdot |T(S)| + d_v \cdot (k_i + k_u) + d_e + e_n + |T(S)| + k_i + k_u)$.

---

$d_e$: “d” for “degenerate” and “e” for “edge”; $e_n$: “e” for “edge” and “n” for “new”.
5.3.8 Combining the Cases

Implementations for the different parts of algorithm 5.1.5 have been given. The cost of these implementations is summarized by lemmas: 5.3.5, 5.3.10, 5.3.13, 5.3.14, and 5.3.20. Instead of combining them to derive a cost for algorithm 5.1.5, we take a different approach, keeping in mind that despite the simplification assumed in section 5.3.2, we are interested in the cost of algorithm 5.1.5 only in so far as it contributes to the cost of algorithm 5.2.1.

Now, part of the cost of algorithm 5.1.5 arises from our use of union-find. After scanning the lemmas referenced above, we see that this cost is

$$O(k_f \cdot |T(S)| + k_i + k_u + k_u \cdot |T(S)| + d_v \cdot (k_i + k_u) + k_i + k_u),$$

where $d_v$ is the total number of non-root marker vertices of degenerate nodes in $T'$. Based on this, we make the following obvious remark:

**Remark 5.3.21.** The total number of find operations required by algorithm 5.1.5 is $O(|T(S)|)$, and the total number of initialization and union operations required is $O(d_v)$, where $d_v$ is the total number of non-root marker vertices of degenerate nodes in $T'$.

Scanning those same lemmas, we realize that the cost of algorithm 5.1.5 not associated with union-find is $O(|T(S)| + d_e + e_n)$, where $d_e$ is the total number of label-edges of degenerate nodes in $T'$, and $e_n$ is the total number of new label edges created by node-joins performed during algorithm 5.1.4. We can therefore state the running-time of algorithm 5.1.5 as follows:

**Lemma 5.3.22.** Algorithm 5.1.5 takes time $O(k + |T(S)| + d_e + e_n)$, where $k$ is the cost of utilizing union-find, and $d_e$ is the total number of label-edges of degenerate nodes in $T'$, and $e_n$ is the total number of new label edges created by node-joins performed during algorithm 5.1.4.

In interpreting the last lemma, recall that the contribution of union-find to the cost of algorithm 5.2.1 is measured in terms of the total number of initialization, union, and find operations (see section 5.3.1). To that end, remark 5.3.21 suffices. Lemma 5.3.22 is intended to capture the costs not associated with union-find.

We will undertake the aggregate analysis required by union-find in the next section. An aggregate approach will also be adopted there for the costs not associated with union-find – and not just for consistency: we will show that an aggregate analysis is necessary to prove that algorithm 5.2.1 (almost) runs in linear-time.

5.4 An Aggregate Analysis

Algorithm 5.2.1 is composed of two parts: the computation of an LBFS for its input graph $G$, and a loop that iteratively adds vertices to the split-tree according to algorithm 5.1.5. The previous section
Figure 5.8: On the left, a path $G$ and its split tree $ST(G)$. On the right, the split tree for $ST(G + (x, S))$, where $S$ consists of the ends of the path.

described an implementation for each of these two parts; the cost associated with algorithm 5.1.5 is summarized by lemma 5.3.22. Unfortunately, extending this lemma in the natural way belies the efficiency of algorithm 5.2.1.

Assume that $G$ has $n$ vertices and $m$ edges. Then the cost of algorithm 5.1.5 is $\Omega(n)$. To see why, consider the example in figure 5.8, where one new vertex adjacent to the ends of a path collapses a split-tree with $\Omega(n)$ nodes into a split-tree with a single node. Following an LBFS ordering does not help because the vertex $x$ in the example could be an end-vertex.

Fortunately, the example in figure 5.8 is worst-case behaviour. This section presents an aggregate analysis of algorithm 5.2.1 to derive an $O(\alpha(n + m) \cdot (n + m) + (n + m))$ bound on its running-time. We do so by bounding the variables in remark 5.3.21 and lemma 5.3.22 as they exist over the course of algorithm 5.2.1. More precisely, we bound the following:

- $D_v$, the total number of non-root marker vertices of degenerate nodes in the tree $T'$ input to algorithm 5.1.4 throughout algorithm 5.2.1;
- $D_e$, the total number of label-edges of degenerate nodes in the set $T'$ input to algorithm 5.1.4 throughout algorithm 5.2.1;
- $N = \Sigma_{i=1}^{n}|T_{i-1}(S_i)|$, where $ST(G_i) = (T_i, F_i)$ and $S_i = N(x_i) \cap V(G_i)$;
- $E_n$, the total number of new label edges created by algorithm 5.1.4 throughout algorithm 5.2.1.

Notice that $D_v < D_e$. Meanwhile, $N$ can be bounded in terms of $E_n$:

**Lemma 5.4.1.** $N$ is $O(n + m + E_n)$.

**Proof.** We use a counting argument. Consider $T_{i-1}(S_i)$, for some $i$. We can partition its nodes into those that are inherited by $ST(G_i)$ and those that are not.
Those nodes in $T_{i-1}(S_i)$ that are inherited by $ST(G_i)$ also belong to $T_i(S_i)$. We know that $|T_i(S_i)|$ is $O(|S_i|)$, by lemma 3.3.19. The total number of nodes of this type throughout algorithm 5.2.1 is therefore $O(m)$.

Those nodes in $T_{i-1}(S_i)$ that are not inherited by $ST(G_i)$ are those in the set $U$ returned by algorithm 5.1.2, by definition. Of course, either $|U| = 1$ or not. The total number of non-inherited nodes throughout algorithm 5.2.1 when $|U| = 1$ is clearly $O(n)$.

When $|U| > 1$, then algorithm 5.1.4 is executed. It follows that every tree edge induced by $U$ is contracted by a node-join. As $U$ induces a tree, by lemma 5.1.8, there are $|U|$ such node-joins. For each one, at least one new label edge is created. Therefore, the total number of non-inherited nodes throughout algorithm 5.2.1 when $|U| > 1$ is $O(E_n)$.

Bounding $D_e$ and $E_n$ is our focus below. We use a charging argument to derive the bound. It is important to note that the charging argument only exists to prove this bound; no part of it is required for the implementation.

### 5.4.1 Setting Up the Charging Argument

The idea for the charging argument is to associate every non-root marker vertex in the GLT with a vertex in the underlying graph. The key part is that if vertex $y$ is associated with marker vertex $q$, and the vertex $z$ is associated with marker vertex $r$, then $y \in A(q)$ and $z \in A(r)$. Thus, if a new label-edge is ever created between $q$ and $r$, then $y$ and $z$ must be adjacent in the underlying graph.

The cost of creating the (hypothetical) new $qr$ marker edge will be charged to the graph edge $yz$. No vertex will be associated with more than two different marker vertices, thereby bounding the charge that will be assigned to any single graph edge. The creation of new degenerate marker vertices can be similarly accounted for.

There are two parts to the charging argument. The first merely counts the total number of new label-edges created and the total number of degenerate marker vertices created. It does so by assigning one unit of charge every time another one is created. This charge resides on Charge lists associated with every marker vertex.

The second part of the charging argument associates vertices in the underlying graph with non-root marker vertices. The charge that exists on the root marker vertices can be “downloaded” to their adjacent non-root marker vertices. Then this charge can be associated with edges in the underlying graph as just explained.

The Charge lists themselves are defined as follows:
**Definition 5.4.2.** Let q be a marker vertex. Then we associate with q two lists of vertices Charge\(_d\)(q) and Charge\(_e\)(q), where:

- the vertices of each list are divided into groups, one for each of q’s neighbours;
- the vertices in neighbour t’s group are the vertices in A(t);
- the group corresponding to the root marker vertex is called the root group (if the root marker vertex is a neighbour of q).

Charge will be assigned to the vertices in the Charge lists. If q is inherited by ST\((G_{i+1})\) from ST\((G_i)\), then it takes its Charge lists with it, along with the charge assigned to them; here we say that the Charge lists have also been inherited. Recall that \(x_{i+1}\) becomes a leaf in ST\((G_{i+1})\); inherited Charge lists are expanded where applicable to include the new vertex \(x_{i+1}\) according to the definition.

If a marker vertex q in ST\((G_i)\) is not inherited by ST\((G_{i+1})\), then the charge stored on its Charge lists will be redistributed to other charge lists (that are inherited by ST\((G_{i+1})\)). The Charge lists assigned to q will then be deleted along with q. As new marker vertices are created, new Charge lists are created for them, and these initially will not hold any charge.

We use charge to calculate \(D_e\) and \(E_n\). In particular, one unit of charge will be assigned to some Charge\(_d\) list for every label-edge of a degenerate node input to algorithm 5.1.4 during algorithm 5.2.1. Similarly, one unit of charge will be assigned to some Charge\(_e\) list for every new label-edge created by a node-join performed by algorithm 5.1.4 during algorithm 5.2.1. No other charge will be assigned. To help prove equality between charge and \(D_e\) and \(E_n\), we introduce the following notation:

**Notation 5.4.3.** We will use \(D_e^i\) to denote the total number of label-edges of degenerate nodes that have been input to algorithm 5.1.4 as part of the tree \(T'\) during the first \(i\) iterations of the loop in algorithm 5.2.1. We will use \(E_n^i\) to denote the total number of new label edges that have been created by node-joins performed in algorithm 5.1.4 during the first \(i\) iterations of the loop in algorithm 5.2.1.

The following invariant is maintained throughout algorithm 5.2.1:

**Invariant 5.4.4.** After the \(i\)th iteration of the loop in algorithm 5.2.1, the total charge assigned to all Charge\(_d\) lists equals \(D_e^i\), and the total charge assigned to all Charge\(_e\) lists equals \(E_n^i\).

To prove this invariant, we will need the slightly stronger invariant below. Its statement reflects the following convention:
Convention 5.4.5. Let \( q \) be a marker vertex. Then \( \text{Charge}(q) \) is meant to stand equally for \( \text{Charge}_d(q) \) and \( \text{Charge}_e(q) \). That is, any statement with respect to \( \text{Charge}(q) \) applies equally to \( \text{Charge}_d(q) \) and \( \text{Charge}_e(q) \).

Invariant 5.4.6. Consider some marker vertex \( q \in V(u) \) in \( ST(G_i) \), as computed by the \( i \text{th} \) iteration of the loop in algorithm 5.2.1. Then:

1. the list \( \text{Charge}(q) \) is free of any charge if any of the following hold:
   
   (a) \( u \) is degenerate;
   
   (b) \( d(q) = 1 \);
   
   (c) \( q \) is a root marker vertex, and there is no \( j < i \) such that \( q \) is inherited from \( ST(G_j) \) and \( q \) is the centre of a star in \( ST(G_j) \);

2. the root group in \( \text{Charge}(q) \) (if it exists) is free of any charge;

3. at most one vertex in each group in \( \text{Charge}(q) \) has been assigned charge;

4. every vertex in \( \text{Charge}(q) \) has been assigned at most 1 unit of charge if \( q \) is a root marker vertex, and at most 3 units of charge otherwise.

In invariant 5.4.6, we implicitly allow for the possibility that \( d(q) = 1 \) when \( u \) is not degenerate. Of course, this is impossible in \( ST(G_i) \), since it must be reduced. Despite this, we allow for the possibility because we will prove invariant 5.4.6 by showing that it holds after each node-join performed during algorithm 5.1.4. During this transitional state, \( u \) may not be degenerate but \( d(q) = 1 \). For example, consider the result of a node-join between a star and some other node. The fact that \( q \)'s \( \text{Charge} \) lists are free of any charge in this case will be necessary below in the proof of invariant 5.4.6.

5.4.2 (Re)Assigning Charge

Consider the \( i \text{th} \) iteration of the loop in algorithm 5.2.1. Assume that invariants 5.4.4, and 5.4.6 both hold after the \( i - 1 \text{st} \) iteration. The split-tree \( ST(G_i) \) is computed from \( ST(G_{i-1}) \) in one of three ways: either algorithm 5.1.1 returns \( ST(G_i) \), algorithm 5.1.3 returns \( ST(G_i) \), or algorithm 5.1.4 returns \( ST(G_i) \).

It is clear from the definitions that in the first two cases \( D_e^i = D_e^{i-1} \) and \( E_n^i = E_n^{i-1} \). Hence, for these two cases, we will not assign any charge. Thus, invariant 5.4.4 continues to hold after the \( i \text{th} \)
iteration. Examining algorithms 5.1.1 and 5.1.3, it is easy to see that invariant 5.4.6 also continues to hold. The key is condition 1(a) stating that the Charge lists for marker vertices in degenerate nodes are free of any charge.

If algorithm 5.1.4 returns \( ST(G_i) \), then cleaning first takes place. Once more, by its condition 1(a), invariant 5.4.6 continues to hold after cleaning. Our focus, therefore, is on algorithm 5.1.4 itself. Obviously, \( D^i_e > D^{i-1}_e \) and \( E^i_n > E^{i-1}_n \) in this case. Below we show how to assign charge to preserve invariants 5.4.4 and 5.4.6 throughout algorithm 5.1.4.

Counting Degenerate Label Edges

We are interested in counting the number of label edges of degenerate nodes in the set input to algorithm 5.1.4. In particular, we will show how charge is assigned and redistributed to/from Charge_d lists for this purpose. The next lemma will help us:

**Lemma 5.4.7.** Suppose that algorithm 5.1.4 is executed while constructing \( ST(G_i) \) from \( ST(G_{i-1}) \) during algorithm 5.2.1. Let \( (T', F') \) be the GLT input to algorithm 5.1.4 for this purpose. Consider some \( q \in V(u) \) such that \( u \) is a node in \( T' \). Assume that \( q \) is inherited by \( ST(G_j), j > i \). If \( q \in V(u') \) in \( ST(G_j) \), then \( u' \) is prime.

**Proof.** A simple inductive argument applying lemma 5.1.33.

Assigning Charge  

Let \( u \) be a degenerate node in the GLT input to algorithm 5.1.4 during the \( i^{th} \) iteration of the loop in algorithm 5.2.1. Recall that invariants 5.4.4 and 5.4.6 are assumed to hold.

Consider some \( q \in V(u) \) such that \( d(q) > 1 \). Choose some \( t \in N(q) \). Then \( t \)'s group in \( Charge_d(q) \) is free of any charge, by condition 1(a) of invariant 5.4.6. So assign one unit of charge to some vertex in that group. This accounts for the label-edge \( qt \). If \( t \) happens to be \( u \)'s root marker vertex, meaning \( t \)'s group in \( Charge_d(q) \) is its root group, then transfer that one unit of charge to a vertex assigned charge in another group, which must exist. By repeating this process for other marker vertices in \( u \), we can account for every one of \( u \)'s label-edges.

At this point, all but condition 1(a) of invariant 5.4.6 continues to hold with respect to \( Charge_d \). If \( q \) is inherited by \( ST(G_i) \), then condition 1(a) will hold in \( ST(G_i) \), by lemma 5.4.7. Furthermore, in that case invariant 5.4.4 will also obviously hold with respect to \( D^i_e \). If \( q \) is not inherited by \( ST(G_i) \), then neither is \( Charge_d(q) \). Therefore, to preserve invariant 5.4.4, we must redistribute the charge just assigned so that it resides on \( Charge_d \) lists that are inherited by \( ST(G_i) \). Of course, we must do so in a manner that is consistent with invariant 5.4.6.
Redistributing Charge  Consider one of the node-joins performed during algorithm 5.1.4. Assume that just before the node-join is performed, all but condition 1(a) of invariant 5.4.6 holds with respect to $\text{Charge}_d$. Let $u$ and $v$ be the nodes that are joined, with $u$ the parent of $v$. Let $q \in V(u)$ and $r \in V(v)$ be the markers of the edge $uv$. Recall the implementation of node-joins from section 5.3.7.

Case 1: Assume the node-join of $u$ and $v$ is a child-extremity node-join. Then $v$ is a star, by lemma 5.3.17; let $c \neq r$ be its centre. In this case, $r$ and $c$ are deleted, and $q$ is repurposed to play the role of $c$ (see figure 5.7). By condition 1(b) of invariant 5.4.6, $\text{Charge}_d(r)$ is free of any charge, and so no charge needs to be redistributed with its deletion. We focus, therefore, on the redistribution of charge on $\text{Charge}_d(c)$.

Let $t \neq r$ be a neighbour of $c$. Observe that $t$ becomes a neighbour of $q$. Therefore a new group is created for $t$ in $\text{Charge}_d(q)$; any charge that was in $t$’s group in $\text{Charge}_d(c)$ can be redistributed there. Since $r$’s group in $\text{Charge}_d(c)$ is $\text{Charge}_d(c)$’s root group, it is free of any charge. Thus, no more charge must be redistributed. Furthermore, notice that no condition of invariant 5.4.6 is violated by this redistribution.

Case 2: Assume the node-join of $u$ and $v$ is a parent-extremity node-join. Then $u$ is a star, by lemma 5.3.17. In this case $q$ and $r$ are both deleted. By condition 1(b) of invariant 5.4.6, $\text{Charge}_d(q)$ is free of any charge, and so no charge needs to be redistributed with its deletion. We focus, therefore, on the redistribution of charge on $\text{Charge}_d(r)$.

The key is that there can be no $j < i$ such that $r$ is inherited from $ST(G_j)$ and $r$ is the centre of a star, by lemma 5.2.11 and the fact that $d(q) = 1$. Therefore, $\text{Charge}_d(r)$ is free of charge, by condition 1(c) of invariant 5.4.6. It follows that there is no charge to redistribute in this case.

Case 3: Assume the node-join of $u$ and $v$ is a regular node-join. Then $d(q), d(r) > 1$ in this case, and both $q$ and $r$ are deleted. Let $t$ be a neighbour of $q$, and let $s$ be a neighbour of $r$. Of course, $t$ becomes universal to $N(r)$ and $s$ becomes universal to $N(q)$. In terms of $\text{Charge}_d(t)$ and $\text{Charge}_d(s)$, we can view this as follows: $q$’s group in $\text{Charge}_d(t)$ is subdivided into $d(r) > 1$ new groups, and $r$’s group in $\text{Charge}_d(s)$ is subdivided into $d(q) > 1$ new groups.

Now, at least one of the new groups in $\text{Charge}_d(t)$ must be free of charge, by condition 3 of invariant 5.4.6. Moreover, none of the new groups becomes $\text{Charge}_d(t)$’s root group. Therefore, whatever charge that resides on $t$’s group in $\text{Charge}_d(q)$ can be moved to one of the new groups in $\text{Charge}_d(t)$ that is free of any charge.

On the other hand, every new group in $\text{Charge}_d(s)$ must be free of charge, by condition 2 of invariant 5.4.6. At least one of these new groups does not become $\text{Charge}_d(s)$’s root group, so whatever charge that resides on $s$’s group in $\text{Charge}_d(r)$ can be reassigned to this non-root group.
free of any charge.

By repeating this process for each pair of neighbours, the necessary charge can be redistributed. Furthermore, notice that no condition of invariant 5.4.6 is violated by this redistribution.

It follows from the three cases above that the charge assigned to $\text{Charge}_d(q)$ for any $q$ that is not inherited by $ST(G_i)$ can be redistributed without violating invariant 5.4.6. As already mentioned, those marker vertices $q$ that are inherited by $ST(G_i)$ satisfy invariant 5.4.6 with respect to $\text{Charge}_d(q)$. Of course, invariant 5.4.4 continues to hold with respect to $D_i$ given the assignment of charge described earlier. It remains to show how charge is assigned so that the two invariants continue to hold with respect to $\text{Charge}_e$ and $E_i$, respectively.

**Counting New Label Edges**

This section shows how charge is assigned and redistributed to/from $\text{Charge}_e$ lists to count the number of new label edges created by node-joins during algorithm 5.1.4’s execution on the $i^{th}$ iteration of the loop in algorithm 5.2.1. Unlike above, we will address the assignment and redistribution of charge together.

Consider one of the node-joins performed during algorithm 5.1.4. Assume that just before the node-join is performed, invariant 5.4.6 holds with respect to $\text{Charge}_e$. Let $u$ and $v$ be the nodes that are joined, with $u$ the parent of $v$. Let $q \in V(u)$ and $r \in V(v)$ be the markers of the edge $uv$. Recall the implementation of node-joins from section 5.3.7.

**Case 1**: Assume the node-join of $u$ and $v$ is a child-extremity node-join. Then $v$ is a star, by lemma 5.3.17; let $c \neq r$ be its centre. In this case, $r$ and $c$ are deleted, and $q$ is repurposed to play the role of $c$ (see figure 5.7). But by condition 1(a) of invariant 5.4.6, both $\text{Charge}_e(r)$ and $\text{Charge}_e(c)$ must be free of charge. Thus, no charge needs to be redistributed in this case.

We focus instead on the new label-edges that are created by the node-join. There are $d(c) - 1$ such label-edges, by remark 5.3.19. Here, $q$ gets $d(c) - 1$ new neighbours, and therefore $\text{Charge}_e(q)$ gets $d(c) - 1$ new groups. One unit of charge can be assigned to each of these groups.

**Case 2**: Assume the node-join of $u$ and $v$ is a parent-extremity node-join. Then $u$ is a star, by lemma 5.3.17. In this case $q$ and $r$ are both deleted. By condition 1(b) of invariant 5.4.6, $\text{Charge}_d(q)$ is free of any charge, and so no charge needs to be redistributed with its deletion. In fact, $\text{Charge}_d(r)$ must also be free of charge, by lemma 5.2.11, and the fact that $d(q) = 1$.

We focus instead on the new label-edges that are created by the node-join. There are $d(r) \cdot d(q) = d(r)$ such label-edges, by remark 5.3.18. Let $c$ be the centre of $v$’s star; clearly $c$ is $v$’s only neighbour. By condition 1(a) of invariant 5.4.6, $\text{Charge}_e(c)$ is free of any charge. In particular, $q$’s group in
Charge_e(c) is free of charge. Since q is deleted and c becomes universal to N(r), it happens that q’s group in Charge_e(c) is subdivided into d(r) new groups, all free of any charge. We assign one unit of charge to each of these groups.

Case 3: Assume the node-join of u and v is a regular node-join. Here we know that d(q), d(r) > 1. There are d(q) · d(r) new label-edges created, by remark 5.3.18. Also, both q and r are deleted as a result of the join.

Let t be a neighbour of q. Then t becomes universal to N(r). In terms of Charge_e(t), we can view this as follows: q’s group in Charge_e(t) is subdivided into d(r) > 1 new groups. Now, all but one of the d(r) > 1 new groups in Charge_e(t) is free of any charge. Redistribute any charge assigned to t’s group in Charge_e(q) to one of these new groups in Charge_e(t) that is free of any charge. By repeating this for all of q’s neighbours, we can redistribute all the charge assigned to Charge_e(q).

Let s be a neighbour of r. In a similar way, s becomes universal to N(q). So r’s group in Charge_e(s) is subdivided into d(q) > 1 new groups. Each of these new groups is free of charge, by condition 3 of invariant 5.4.6. Whatever charge is on s’s group in Charge_e(r) can be redistributed to one of these groups. That leaves d(q) – 1 groups free of charge. One of these groups can receive two units of charge, and each of the others can receive one unit of charge, to account for s’s d(q) new incident edges. At most one of these new groups can correspond to Charge_e(s)’s root group. If so, then the charge assigned to it can be move to another. By repeating this for all of r’s neighbours, we can redistribute all the charge assigned to Charge_e(r), and account for the d(q) · d(r) new label edges created.

It is a simple (but tedious) exercise to verify that invariant 5.4.6 continues to hold after the (re)assignment of charge just described. Focusing on condition 1(a), for example, we remark that the node-join of u and v cannot be degenerate; hence, condition 1(a) continues to hold. Obviously, invariant 5.4.4 also continues to hold. Combining this fact with the observations in section 5.4.2, we conclude the following:

Lemma 5.4.8. Invariants 5.4.4 and 5.4.6 both hold at the end of algorithm 5.2.1.

5.4.3 Bounding the Total Charge

We now quantify the total charged assigned by our charging argument. Consider all Charge lists as they exist after algorithm 5.2.1. Bounding all the charge residing on these Charge lists is a three-step process: first we “download” all charge from root marker vertices to non-root marker vertices; then we assign stamps to each non-root marker vertex; finally, we use the stamps to
associate charge with edges in the underlying graph. The total charge associated with each edge will be constant. The total charge overall must then be $O(n + m)$. In the details that follow below, we continue to follow convention 5.4.5.

**Downloading Charge**

We are interested in removing any charge that resides on $\text{Charge}$ lists of root marker vertices. We can assume that invariant 5.4.6 holds, by lemma 5.4.8. Therefore, we can concentrate on root marker vertices of prime nodes, since marker vertices of degenerate nodes are free of charge, by condition 1(a) of invariant 5.4.6.

Let $r$ be the root marker vertex of a prime node $u$, and let $s$ be one of $r$’s neighbours. By conditions 3 and 4 of invariant 5.4.6, at most one vertex in $s$’s group in $\text{Charge}(r)$ is assigned charge, and the charge assigned to that vertex is one. On the other hand, $r$’s group in $\text{Charge}(s)$ is free of any charge, by condition 2 of invariant 5.4.6. So we redistribute the charge from $s$’s group in $\text{Charge}(r)$ to $r$’s group in $\text{Charge}(s)$. We do this for every other neighbour of $r$ as well, and call the process downloading charge from $r$. The obvious result is summarized in the next remark:

**Remark 5.4.9.** The following holds after downloading all charge from root marker vertices of prime nodes:

1. if $r$ is a root marker vertex, then $\text{Charge}(r)$ is free of any charge;

2. if $q$ is a non-root marker vertex of a degenerate node, then $\text{Charge}(r)$ is free of any charge;

3. if $q$ is a non-root marker vertex of a prime node, then at most one vertex in any one of its groups has been assigned charge, and that vertex has not been assigned more than three units of charge.

**Stamping Non-Root Marker Vertices**

We will assign a pair of stamps to each non-root marker vertex (that is not the centre of a star). If $q$ is such a marker vertex, then its pair of stamps will be denoted $s(q) = (s_1(q), s_2(q))$. We call $s_1(q)$ the primary stamp and $s_2(q)$ the secondary stamp. Consider the procedure below for assigning these stamps:

1. If $q$ is opposite the leaf $y$, then $s(q) = (y, y)$.

2. Let $uv$ be an internal tree edge in the split-tree $ST(G)$ with extremities $q \in V(u)$ and $r \in V(v)$, where $u$ is the parent of $v$: 

112
(a) if $d(r) > 1$, then set $s(q) = (s_2(t), s_2(t'))$ for two (arbitrary) neighbours $t$ and $t'$ of $r$;

(b) if $d(r) = 1$, and therefore $v$ is a star with centre $c$, then set $s(q) = s(c)$, and then remove $c$’s stamp.

Notice that only step 2(b) removes a stamp. Therefore, at the end of the procedure, every non-root marker vertex that is not the centre of a star has a stamp. In fact, we are only interested in primary stamps; secondary stamps only exist to help assign primary stamps in the above procedure. Concerning primary stamps, we observe the following two results:

**Lemma 5.4.10.** At the end of the procedure, if the leaf $y$ is the primary stamp of the marker vertex $q$, then $y \in A(q)$.

*Proof.* An easy inductive argument shows this, applying the fact that $q$ only receives a stamp via its neighbours.

**Lemma 5.4.11.** At the end of the procedure, every leaf is a primary stamp at most twice.

*Proof.* Let $uv$ be an arbitrary edge in $T$, where $u$ is the parent of $v$, and let $q \in V(u)$ and $p \in V(v)$ be arbitrary non-root marker vertices, where $p$ is accessible from $q$. We let $y$ be an arbitrary vertex and examine how occurrences of $y$ in $s(p)$ can be transmitted to $s(q)$. Note that the stamp $(y, y)$ applies to the marker vertex opposite $y$, and thus a bottom-up argument starts with $y$ having appeared once as a primary stamp.

First, we observe that no step of the algorithm allows a primary occurrence of $y$ in $s(p)$ to be a secondary occurrence of $y$ in $s(q)$. Suppose for contradiction that a primary occurrence of $y$ in $s(p)$ is also a primary occurrence of $y$ in $s(q)$. This can only happen by execution of step 2(b), but now the stamp is removed from $p$. Thus this case does not allow an increase in the number of times that $x$ appears as a primary stamp.

Finally, suppose that a secondary occurrence of $y$ in $s(p)$ becomes a primary occurrence of $y$ in $s(q)$. Step 2(a) allows this to happen thereby increasing by one the number of times that $y$ can appear as a primary stamp. The preceding argument shows that this cannot occur again.

**Associating Charge with Graph Edges**

The previous two lemmas will allow us to associate charge with edges in the underlying graph. Let $q$ be a non-root marker vertex of a prime node, and let $t$ be one of $q$’s neighbours ($t$ is possibly the root marker vertex). Say $y$ is the vertex in $t$’s group in $\text{Charge}(q)$ that has been assigned charge (if any charge has been assigned to the group). Assume $s_1(q) = x$. Now, by construction, $y \in A(t)$; and by lemma 5.4.10, $x \in A(q)$. Therefore $xy \in E(G)$.
We associate the charge residing on \( y \) in \( \text{Charge}(q) \) with the edge \( xy \). The total charge thus associated is at most six, by remark 5.4.9 (3 units of charge each for \( \text{Charge}_d(q) \) and \( \text{Charge}_e(q) \)). But by lemma 5.4.11, \( x \) acts as a primary stamp at most twice. Therefore, each edge incident to \( x \) has at most twelve units of charge associated with it. Hence:

**Lemma 5.4.12.** The total charge assigned by our charging argument is \( O(n + m) \).

**Corollary 5.4.13.** \( D_e \) and \( E_n \) are both \( O(n + m) \).

*Proof.* By lemma 5.4.8, we can assume that invariant 5.4.4 holds at the end of algorithm 5.2.1. The result now follows from lemma 5.4.12. \( \square \)

The previous corollary allows us to refine our earlier bound on \( N \):

**Lemma 5.4.14.** \( N \) is \( O(n + m) \).

*Proof.* Immediate from corollary 5.4.13 and lemma 5.4.1. \( \square \)

We will need lemma 5.4.14 in the next chapter, where we extend algorithm 5.2.1 to recognize circle graphs. For the time being, we apply corollary 5.4.13 to conclude the running-time of algorithm 5.2.1.

### 5.4.4 Bounding the Running-Time

Our analysis of the running-time of algorithm 5.2.1 first considers its cost not due to union-find. For that, we extend lemma 5.3.22 in the natural way. The total cost not associated with union-find is therefore \( O(N + D_e + E_n) \). But that is \( O(n + m) \), by combining lemma 5.4.1 and corollary 5.4.13.

The cost associated with union-find can be similarly determined, but this time we extend remark 5.3.21. The total number of *initialization* and *union* operations is therefore \( O(D_v) \), which is \( O(n + m) \), by corollary 5.4.13. Similarly, the total number of *find* operations is \( O(N) \), which is \( O(n + m) \), by combining lemma 5.4.1 and corollary 5.4.13. Overall, we can summarize the running time of algorithm 5.2.1 as follows:

**Theorem 5.4.15.** The running-time of algorithm 5.2.1 is \( O(\alpha(n + m) \cdot (n + m) + n + m) \), where \( \alpha \) is the inverse of Ackermann’s function.

The previous theorem is the main result of this chapter. It proves algorithm’s 5.2.1 efficiency: \( \alpha \) grows so slowly as to be effectively constant for all realistic inputs. Each of the algorithm’s constituent parts is clearly stated and proved, and together they combine to present an algorithm that is straightforward in its understanding. The implementation we have described is practical in
requiring no more than rooted trees and union-find. The difficult part associated with the algorithm has been the proof of its running-time, which required a technical and detailed charging argument, of which theorem 5.4.15 is the culmination. In the next chapter, we use algorithm 5.2.1 to improve the running-time of circle graph recognition.
Chapter 6

(Almost) Linear-Time Circle Graph Recognition

The connection between circle graph recognition and split decomposition was observed in chapter 2. Unfortunately, as remarked there, it is unclear whether the two existing linear-time split decomposition algorithms [55, 27] can be extended to recognize circle graphs. The quadratic bound on circle graph recognition [126] has stood for fifteen years.

This chapter extends our LBFS-incremental split-tree algorithm to be the first to recognize circle graphs in sub-quadratic time. We derive a new LBFS characterization of circle graphs and then translate it to the incremental split-tree setting described in the previous chapter. The efficient implementation of this translation is made possible by a reversal-invariant data-structure we design for realizers. We encourage the reader to revisit chapter 3 to review the convention we adopt for subscripts in double occurrence words. An understanding of this will improve the reading of the proofs that follow.

We will assume that the input graph is connected. Disconnected circle graphs can be recognized by recognizing the graphs induced by their constituent components, which can be identified in linear-time by breadth-first search. To be consistent with the incremental split-tree algorithm of the previous chapter, we further assume that each vertex added in our incremental setting has a non-empty neighbourhood.

This chapter is based on the technical report [80], jointly developed with Derek Corneil, Emeric Gioan, and Christophe Paul.
6.1 Adding a Good Vertex

We are interested here in the problem of adding an end-vertex to an existing circle graph. The following will be the basis of our discussion:

**Definition 6.1.1.** Let \( \pi \) be a realizer for the circle graph \( G \). Then the set \( S \subseteq V(G) \) is consecutive in \( \pi = \cdots \alpha \cdots \) if \( |S| = |\alpha| \) and \( x \in S \) implies \( x \in \alpha \). In this case \( \alpha \) is said to certify that \( S \) is consecutive in \( \pi \).

It is worth underscoring that \( S \) being consecutive means that exactly one occurrence of each \( x \in S \) is contained in any string certifying it is consecutive. Thus, if a circle graph \( G \) has a realizer in which the set \( S \subseteq V(G) \) is consecutive, then \( G + (x, S) \) is clearly a circle graph, as demonstrated in figure 6.1. We characterize the resulting realizer as follows:

**Definition 6.1.2.** Let \( \pi \) be a realizer for the circle graph \( G \), and consider some \( x \in V(G) \). We say that \( x \) encloses \( N(x) \) in \( \pi \) if there exists a substring \( x\alpha x \) of \( \pi \) such that \( \alpha \) certifies that \( N(x) \) is consecutive in \( \pi \). In this case \( \alpha \) is said to certify that \( x \) encloses \( N(x) \).

It turns out that such a realizer always exists if \( x \) is an end-vertex vertex for \( G + (x, S) \). We start by proving this for circle graphs without (non-trivial) modules and then generalize the result to all circle graphs:

**Lemma 6.1.3.** Let \( G \) be a circle graph for which \( x \in V(G) \) is an end-vertex. Assume that \( G \) does not contain a non-trivial module. Then there is a realizer for \( G \) in which \( x \) encloses \( N(x) \).

**Proof.** Let \( \sigma \) be an LBFS of \( G \) in which \( x \) appears last, and suppose that \( s \) is the first vertex in \( \sigma \). Assume for contradiction that no realizer for \( G \) exists in which \( x \) encloses \( N(x) \). Let \( \pi = \alpha x \beta x \gamma \) be a realizer for \( G \).
First consider the case where \( s \) and \( x \) are adjacent. Of course, \( N(s) \) is the second slice encountered during algorithm 3.5.1’s production of \( \sigma \) (the first is \( V(G) \)). So by remark 3.5.3, \( N(s) \) appears consecutively in \( \sigma \). As \( x \) is an end-vertex, this means \( s \) is universal in \( G \). Hence, \( V(G) - \{s\} \) is a module in \( G \). As \( G \) has no non-trivial modules, we must have \( V(G) - \{s\} = \{x\} \); in other words \( |V(G)| = 2 \). The lemma holds trivially in this case.

So assume that \( s \) and \( x \) are not adjacent. We will assume without loss of generality that \( s_1 \notin \beta \) and \( s_2 \notin \beta \). Moreover, since \( x \) does not enclose \( N(x) \) in \( \pi \), we know that there is a vertex \( y \) such that \( y_1, y_2 \in \beta \). Without loss of generality, amongst all such vertices, choose \( y \) to be the one appearing earliest in \( \sigma \).

Let \( B \) be the set of vertices appearing before \( y \) in \( \sigma \). By choice of \( y \), if \( z \in B \) and \( z \in \beta \), then \( z \notin \beta \). Thus, \( N(y) \cap B \subseteq N(x) \cap B \). Hence, \( x \in S(y) \). Since \( x \) is the last vertex in \( \sigma \), the slice \( S(y) \) must be a module. Of course, \( x, y \in S(y) \) and \( x \neq y \), meaning \( |S(y)| > 1 \). Moreover, \( s \notin S(y) \), since \( y \neq s \), by choice of \( s \) and \( y \). Therefore \( S(y) \) is a non-trivial module in \( G \), a contradiction.

To generalize the previous result, we will need the following:

**Corollary 6.1.4.** Let \( G \) be a circle graph for which \( x \in V(G) \) is an end-vertex, and assume \( |V(G)| > 2 \). Let \( \sigma \) be an LBFS certifying that \( x \) is an end-vertex for \( G \). Assume that \( G \) has a realizer in which \( x \) does not enclose \( N(x) \). Then there is a \( k > 1 \) such that \( G \) contains a non-trivial module \( M \) consisting of the last \( k \) vertices (including \( x \)) in \( \sigma \).

**Proof.** Let \( s \) be the first vertex in \( \sigma \). If \( x \) and \( s \) are adjacent, then \( V(G) - \{s\} \) is the desired module, as in the proof of lemma 6.1.3. If \( x \) and \( s \) are not adjacent, then the set \( S(y) \) in the proof of lemma 6.1.3 is such a module.

**Lemma 6.1.5.** Let \( G \) be a circle graph for which \( x \in V(G) \) is an end-vertex. Then there is a realizer for \( G \) in which \( x \) encloses \( N(x) \).

**Proof.** If \( |V(G)| \leq 2 \), then the result follows trivially. So assume that \( |V(G)| = n > 2 \); the proof will be by induction on \( n \). Assume that the lemma holds for all graphs with fewer than \( n \) vertices.

Let \( \sigma \) be an LBFS of \( G \) in which \( x \) appears last. Assume for contradiction that there is no realizer for \( G \) in which \( x \) encloses \( N(x) \).

Let \( M \) be a non-trivial module containing \( x \); we know that such an \( M \) exists, by corollary 6.1.4. Recall that \( G \) is connected, by assumption throughout this chapter. Observe that \( \sigma[M] \) is an LBFS of \( G[M] \), by lemma 3.5.5, and obviously \( x \) appears last in this ordering. In other words, \( x \) is an end-vertex for \( G[M] \). Moreover, \( |M| < n \), since \( M \) is non-trivial. Therefore, by our induction
hypothesis, there is a realizer \( \pi_m = x\alpha x\beta \) for \( G[M] \) in which \( x \) encloses \( N(x) \cap M \). We will assume, without loss of generality, that \( \alpha \) certifies that \( x \) encloses \( N(x) \cap M \).

For simplicity, let \( G' = G[(V(G) - M) \cup \{x\}] \). Then \( |V(G')| < n \), since \( M \) is non-trivial. Moreover, by lemma 3.5.6, we know that \( \sigma[V(G')] \) is an LBFS of \( G' \), and obviously \( x \) appears last in this ordering, meaning \( x \) is an end-vertex for \( G' \). So by our induction hypothesis, there is a realizer \( \pi' = x\alpha' x\beta' \) in which \( x \) encloses \( N[x] \cap V(G') \). We will assume, without loss of generality, that some prefix of \( \alpha' \) certifies that \( x \) encloses \( N[x] \cap V(G') \). But then \( \pi = x\alpha\alpha' x\beta\beta' \) is a realizer for \( G \) in which \( x \) encloses \( N(x) \), a contradiction.

As already demonstrated in figure 6.1, the converse of the previous lemma also holds, providing the following LBFS-incremental characterization of circle graphs:

**Lemma 6.1.6.** Let \( G \) be a graph for which \( x \in V(G) \) is an end-vertex. Then \( G \) is a circle graph if and only if there is a realizer for \( G - x \) in which \( N(x) \) appears consecutively.

**Proof.** Assume that \( G \) is a circle graph. Then by lemma 6.1.5, there is a realizer for \( G \) in which \( x \) encloses \( N(x) \). Removing both occurrences of \( x \) from this realizer proves sufficiency. So assume that \( G - x \) has a realizer \( \pi \) in which \( N(x) \) appears consecutively, and say \( \pi = \alpha x\beta x\gamma \), where \( \beta \) certifies that \( N(x) \) is consecutive. Then \( \alpha x\beta x\gamma \) is a realizer for \( G \) in which \( N(x) \) appears consecutively.

Of course, the previous lemma immediately suggests an LBFS-incremental circle graph recognition algorithm: build the graph one vertex at a time, adding vertices according to an LBFS ordering; for each new vertex \( x \), determine if the graph so far built has a realizer in which \( N(x) \) is consecutive; if so, then continue building the graph, otherwise the input graph cannot be circle. The difficulty lies in determining if such a realizer exists. We show below that GLTs simplify the task.

### 6.2 Consecutivity and GLTs

Let \( G \) be a circle graph and consider some set \( S \subseteq V(G) \). At first glance, the task of determining if \( G \) has a realizer in which \( S \) is consecutive seems difficult. However, given \( ST(G) \), it is actually straightforward. Below we use \( ST(G) \) to define the conditions under which \( G \) has a realizer in which \( S \) is consecutive:

**Definition 6.2.1.** Let \( G \) be a circle graph and let \( \pi \) be a realizer for \( G \) in which the set \( S \subseteq V(G) \) is consecutive. Assume that \( \alpha \) certifies that \( S \) is consecutive in \( \pi \). If \( x \in S \) is the first or last letter in \( \alpha \), then \( x \) is a bookend for \( S \) with respect to \( \alpha \).
Definition 6.2.2. Let $u$ be a node in a GLT $(T, F)$ whose marker vertices have been assigned states with respect to a subset $S$ of $T$’s leaves. Then $u$ satisfies the consecutive constraint if there is a realizer $\pi = \cdots \alpha \cdots$ for $G(u)$ such that:

1. $\alpha$ certifies that $NE(u)$ is consecutive in $\pi$;
2. $q \in M(u)$ implies that $q$ is a bookend with respect to $\alpha$.

In this case $\pi$ is said to certify that $u$ satisfies the consecutive constraint.

The following is obvious of all nodes satisfying the consecutive constraint:

Remark 6.2.3. If $u$ satisfies the consecutive constraint, then $|M(u)| \not\geq 2$.

We go further below, showing that if all nodes in $ST(G)$ satisfy the consecutive constraint, then $G$ has a realizer in which $S$ is consecutive. We then show the converse. It is worth noting that these results assume that all nodes in $ST(G)$ are fully-mixed. Our ultimate goal is to incorporate the results below into an incremental circle graph recognition algorithm based on our LBFS-incremental split-tree algorithm. The latter was reduced to the problem of split-trees consisting of a single unmixed node, or those whose nodes were all fully-mixed. Of course, a single node is vacuously fully-mixed. Hence the fully-mixed assumption made below for $ST(G)$.

6.2.1 Consecutivity in the Labels

Assume that every node in $ST(G)$ is fully-mixed, and every one of its nodes also satisfies the consecutive constraint with respect to $S$. We will prove that $G$ has a realizer in which $S$ is consecutive. The idea is to recursively apply node-joins to arrive at a single node labelled by a graph isomorphic to $G$. We will maintain the invariant that every node in each intermediate GLT in this process satisfies the consecutive constraint. We begin by showing how to form realizers from neighbourhood-joins:

Lemma 6.2.4. Let $G_x$ and $G_y$ be two circle graphs, and assume that $x \in V(G_x)$ and $y \in V(G_y)$. Assume further that $\pi_x = x\alpha x\beta$ is a realizer for $G_x$ and $\pi_y = y\gamma y\delta$ is a realizer for $G_y$. Let $G$ be the graph formed by the neighbourhood-join of $G_x$ and $G_y$ with respect to $x$ and $y$. Then each of the following is a realizer for $G$:

$$
\gamma \alpha \delta \beta \quad \gamma^r \alpha \delta^r \beta \quad \gamma \alpha^r \delta^r \beta^r \\
\delta \alpha \gamma \beta \quad \delta^r \alpha \gamma^r \beta \quad \delta \alpha^r \gamma^r \beta^r
$$

(6.1)
Proof. Easily verified from the definitions.

Next, we apply the previous result to show how consecutivity is maintained after a node-join:

Lemma 6.2.5. Let $G_x$ and $G_y$ be two circle graphs, and assume that $x \in V(G_x)$ and $y \in V(G_y)$. Let $\pi_x$ and $\pi_y$ be realizers for $G_x$ and $G_y$, respectively, and let $S_x \subseteq V(G_x)$ and $S_y \subseteq V(G_y)$ be such that $x \notin S_x$ and $y \notin S_y$. Assume that $S_x \cup \{x\}$ is consecutive in $\pi_x$, and $S_y \cup \{y\}$ is consecutive in $\pi_y$, with $x$ and $y$ bookends, respectively. Assume further that $x' \in S_x$ and $y' \in S_y$ are the other bookends, respectively. Let $G$ be the graph formed by the neighbourhood-join of $G_x$ and $G_y$ with respect to $x$ and $y$. Then there is a realizer for $G$ in which $S_x \cup S_y$ is consecutive and $x'$ and $y'$ are the bookends.

Proof. Assume that $\pi_x = x\alpha x\beta$ and $\pi_y = y\gamma y\delta$. Since $x$ is a bookend, then either some prefix or suffix of $\alpha$ or $\beta$ certifies that $S_x$ is consecutive in $\pi_x$. Similarly, since $y$ is a bookend, then either some prefix of $\gamma$ or $\delta$ or some suffix of $\gamma$ or $\delta$ certifies that $S_y$ is consecutive in $\pi_y$. It is an easy exercise to verify in each case that $S_x \cup S_y$ is consecutive in exactly one of the realizers for $G$ in (6.1) (see lemma 6.2.4), and when consecutive, that $x'$ and $y'$ must be the bookends.

To apply the previous lemma, we first assume that each node in $ST(G)$ is fully-mixed and satisfies the consecutive constraint. Then we recursively apply node-joins until we arrive at a single node labelled by a graph isomorphic to $G$. The previous lemma says that every intermediate GLT in this recursive process will also be such that all of its nodes satisfy the consecutive constraint; in particular, the final node will. This, of course, implies that $G$ has a realizer in which $N(x)$ is consecutive. We formalize this in algorithm 6.2.1 below, and its proof of correctness.

Corollary 6.2.6. Algorithm 6.2.1 is correct.

Proof. Since $ST(G)$ is fully-mixed, the $q$ and $r$ in the loop must be mixed, and therefore bookends, since every node in $ST(G)$ satisfies the consecutive constraint. Every $v$ in the loop satisfies the consecutive constraint, by lemma 6.2.5, with $q$ and $r$ playing the role of $x$ and $y$ in the statement of the lemma. Moreover, the proof of lemma 6.2.5 guarantees that a realizer certifying that $v$ satisfies the consecutive constraint will be one amongst those in (6.1) (see lemma 6.2.4). Hence, the $\pi$ in the second loop correctly certifies that $v$ satisfies the consecutive constraint. It follows inductively that $T$’s only node at the end of the second loop will satisfy the consecutive constraint.

Of course, the single node $u$ after the loop is such that $G(u)$ is isomorphic to $G$. Moreover, every neighbour of $u$ is a leaf, meaning $u$ is unmixed, and the set of marker vertices opposite leaves in $S$ is $P(u)$. But $u$ being unmixed means $NE(u) = P(u)$. Therefore, the string returned by algorithm 6.2.1 is a realizer for a graph isomorphic to $G + (x, S)$. □
Algorithm 6.2.1: *ConsecContractAdd*(ST(G), R, x)

**Input:** The split-tree ST(G), all of whose nodes are fully-mixed and satisfy the consecutive constraint with respect to a non-empty subset S of its leaves, and a set of realizers R certifying this fact; a vertex x not amongst its leaves.

**Output:** A realizer for G + (x, S).

(T, F) ← ST(G);

while T contains an internal tree edge e = uu’ with markers q and r do

let π_u, π_u’ ∈ R be such that π_u = qαqβ and π_u’ = rγrδ;

update (T, F) by node-joining u and u’, and let v be the node that replaces them;

let π be the realizer below certifying that v satisfies the consecutive constraint:

\[
\gamma\delta\alpha\beta \quad \gamma^r\alpha^r\delta^r\beta^r \\
\delta\alpha\gamma\beta \quad \delta^r\alpha^r\gamma^r\beta^r 
\]

R ← (R ∪ {π}) – {π_u, π_u’};

endw

let u be the only node in T, and assume that R = {π}, where π is a realizer for G(u);

assume that π = αβγ, such that β certifies that P(u) is consecutive;

return αxβxγ;
Figure 6.2: On the right, a chord diagram that is partitionable with respect to the split \((A, B)\) on the left.

There is a type of inverse to algorithm 6.2.1, in which one starts with a realizer for \(G\) in which \(S\) is consecutive, and then performs the inverse node-split operations. The result will be a split-tree in which every node satisfies the consecutive constraint. This fact is formally proved below.

6.2.2 Consecutivity in the Graph

We explore here what implications there are for \(ST(G)\) when \(G\) has a realizer in which \(S\) is consecutive, and every node in \(ST(G)\) is fully-mixed. First we need to show how splits can partition realizers:

**Definition 6.2.7.** Let \(\pi\) be a realizer for a circle graph \(G\) containing the split \((A, B)\). Then \(\pi\) is partitionable with respect to \((A, B)\) if \(\pi = \alpha_1\beta_1\alpha_2\beta_2\) such that \(\alpha_i, \beta_i \neq \epsilon\), and \(x \in \alpha_i\) implies \(x \in A\), and \(y \in \beta_i\) implies \(y \in B\).

**Remark 6.2.8.** Let \(\pi = \alpha_1\beta_1\alpha_2\beta_2\) be partitionable with respect to the split \((A, B)\), and let \(A'\) and \(B'\) be the frontiers of the split. Then \(x \in A'\) if and only if \(x_1 \in \alpha_1\) and \(x_2 \in \alpha_2\); and \(y \in B'\) if and only if \(y_1 \in \beta_1\) and \(y_2 \in \beta_2\).

Partitionability can be visualized as in figure 6.2. Its relation to the realizers in (6.1) (see lemma 6.2.4) is not difficult to see. This duality is primarily responsible for the value split decomposition has for circle graphs, including the uniqueness of prime realizers (see lemma 3.4.5).

Unfortunately, not all splits allow for this type of partitioning. Recall from theorem 3.3.17 the two ways in which the split-tree can encode a split. We see in figure 6.3 that those splits encoded by degenerate nodes do not always allow for this type of partitioning. However, the next lemma proves the opposite for splits encoded by internal tree-edges in the split-tree:
Lemma 6.2.9. Let $q$ and $r$ be the markers of an internal tree edge in $ST(G)$, where $G$ is a circle graph. Then any realizer for $G$ is partitionable with respect to the split $(L(q), L(r))$.

Proof. By remark 3.3.12, $(L(q), L(r))$ is a split in $G$ with frontiers $A(q)$ and $A(r)$. Suppose that $q \in V(u)$ and $r \in V(v)$. Assume that $\pi = \alpha_1 \beta_1 \ldots \alpha_k \beta_k$, where $\alpha_i, \beta_i \neq \epsilon$, and $x \in \alpha_i$ implies $x \in L(q)$, and $y \in \beta_i$ implies $y \in L(r)$. Assume for contradiction that $k > 2$. We contradict this by a series of claims below. The first seven are stated with respect to $\alpha$, but hold equally for $\beta$, by symmetry. Similarly, the last two claims are stated with respect to $v$ but hold equally for $u$, by symmetry.

Claim 1: There is no $\alpha_i$ such that $a_1, a_2 \in \alpha_i$ for all $a \in \alpha_i$. If not, then $G$ would be disconnected.

Claim 2: If $a_1 \in \alpha_i$ and $a_2 \in \alpha_j, i \neq j$, then $a \in A(q)$. Assume $a \in L(q) - A(q)$ for contradiction. Without loss of generality, assume $i < j$. We know there exists a $\beta_\ell, i \leq \ell \leq j$; and there exists a $\beta_{\ell'}, j \leq \ell' \leq k$. Observe that there can be no $b \in L(r)$ such that $b_1 \in \beta_\ell$ and $b_2 \in \beta_{\ell'}$, since then $a$ and $b$ would be adjacent, contradicting $a \in L(q) - A(q)$. So if $b, b' \in L(r)$ are such that $b \in \beta_\ell$ and $b' \in \beta_{\ell'}$, then there can be no path between $b$ and $b'$ consisting entirely of vertices in $L(r)$, which contradicts $G$ being connected.

Claim 3: The converse of claim 2 holds: if $a \in A(q)$, then there exists $i \neq j$ such that $a_1 \in \alpha_i$ and $a_2 \in \alpha_j$. Otherwise such an $a \in A(q)$ (resp. $b \in A(r)$) could not be adjacent to a $b \in A(r)$ (resp. $a \in A(q)$), meaning $G$ would be disconnected.

Claim 4: For each $\alpha_i$, there is some $a \in A(q)$ such that $a \in \alpha_i$. We get this by combining claims 1 and 2.

Claim 5: If $a, a' \in A(q)$, $a_1, a'_1 \in \alpha_\ell$, and $a_2, a'_2 \in \alpha_j$, then $i = j \neq \ell$. First, notice that $i, j \neq \ell$, by claim 3. Suppose for contradiction that $i \neq j$; without loss of generality, assume...
\( i < j \). By claim 4, we know that there exists a \( b \in A(r) \) such that \( b_1 \in \beta_i \). Of course, \( b \) is adjacent to both \( a \) and \( a' \), since \( b \in A(r) \). But this forces \( b_2 \in \alpha_{\ell} \), a contradiction.

**Claim 6:** If \( a, a' \in A(q) \) are non-adjacent, then there exists \( i \neq j \) such that \( a_1, a'_1 \in \alpha_i \) and \( a_2, a'_2 \in \alpha_j \). The argument is similar to that in claim 5.

**Claim 7:** There exists an \( \alpha_i \) and adjacent vertices \( a, a' \in A(q) \) such that \( a \in \alpha_i, a' \notin \alpha_i \). Consider \( \alpha_1, \alpha_2, \) and \( \alpha_3 \). For each of them, we can associate a vertex in \( A(q) \), by claim 4. At least two of these three vertices must be distinct, by claim 3. The two distinct vertices must be adjacent, by claim 6. Finally, we have \( i \in \{1, 2, 3\} \).

**Claim 8:** If \( v \) is a star, then \( r \) is its centre. Assume for contradiction that \( v \) is a star whose centre is \( c \) and \( r \neq c \). Let \( t \in V(v) \) be an extremity different from \( r \). Choose some \( d \in A(t) \). Notice that \( d \in L(q) - A(q) \). We can assume without loss of generality that \( d_1 \in \alpha_1 \). Then by claim 2, we know that \( d_2 \in \alpha_1 \) as well. Notice that \( d \) must be universal to \( A(c) = A(q) \). Therefore, for each \( a \in A(q) \), we have \( a_1 \in \alpha_1 \). Furthermore, by claim 3, \( a_2 \in \alpha_i \), for some \( i > 1 \). If \( A(q) = \{a\} \), then obviously \( k = 2 \). So assume that there is some \( a' \in A(q) \) such that \( a' \neq a \). Then as for \( a \), we have \( a_1' \in \alpha_1 \); and by claim 5, we must have \( a_2' \in \alpha_i \). Thus, \( k = 2 \), by claim 4.

**Claim 9:** The node \( v \) is not a star. Assume for contradiction that \( v \) is a star. By claim 8, \( r \) must be the centre of \( v \). Let \( t \in V(v) \) and \( s \in V(v) \) be distinct extremities of \( v \)'s star. Choose some \( a \in A(t) \) and some \( a' \in A(s) \). Observe that \( a, a' \in A(q) \) and that \( a \) and \( a' \) are not adjacent. So by claim 6, there are \( i \neq j \) such that \( a_1, a'_1 \in \alpha_i \) and \( a_2, a'_2 \in \alpha_j \). As \( k > 2 \), there exists \( \alpha_{\ell}, \ell \neq i, j \). By claim 4, there is some \( a'' \in A(q) \) such that \( a'' \in \alpha_{\ell} \). Since \( t \) and \( s \) are distinct, \( a'' \) cannot simultaneously be adjacent to \( a \) and \( a' \). But then by claim 6, either \( \ell = i \) or \( \ell = j \), a contradiction.

The preceding claims will now be used to derive the necessary contradiction. Notice that one of \( u \) and \( v \) must be prime, since neither can be a star, by claim 9, and both cannot be cliques, since \( ST(G) \) is reduced. So assume without loss of generality that \( u \) is prime.

Let \( a, a' \in A(q) \) be the adjacent pair guaranteed by claim 7. The claim guarantees that there is an \( \alpha_i \) such that \( a \in \alpha_i, a' \notin \alpha_i \). Let \( \pi = a \gamma a' \gamma' a \delta a' \delta' \) be a realizer for \( G \). Then at most one of \( \gamma, \gamma', \delta, \delta' \) does not contain a \( \beta_j \) as a substring. So assume without loss of generality that \( \gamma \) and \( \gamma' \) both contain a \( \beta_j \) as a substring. By claim 4, there are vertices \( b, b' \in A(r) \) such that \( b \in \gamma \) and \( b' \in \gamma' \) (possibly \( b = b' \)). Let \( s, s' \in V(u) \) be such that \( b \in A(s) \) and \( b' \in A(s') \) (possibly \( s = s' \)).

For each \( t \in V(u) - \{s, s', q\} \), associate a vertex in \( A(t) \); let \( S \) be the set of vertices thus chosen. Consider the double occurrence words \( \pi_u = \pi[S \cup \{b, b', a\}] \) and \( \pi'_u = \pi[S \cup \{b, b', a'\}] \). Then \( \pi_u = a \cdots b \cdots b' \cdots a' \cdots a \cdots \) and \( \pi'_u = a \cdots b \cdots a' \cdots b' \cdots a' \cdots \). Obviously, both \( \pi_u \) and \( \pi'_u \) are realizers for a graph isomorphic to \( G(u) \). Furthermore, relabelling \( a' \) as \( a \) means both have the same letters. Nevertheless, if \( \tau \) is a rotation of \( \pi_u \), then \( \tau \neq \pi'_u \) and \( \tau^r \neq \pi'_u \). This contradicts the uniqueness of
prime realizers (lemma 3.4.5).

The next result is a sort of converse of lemma 6.2.5. In the same way the latter contributed to the proof of corollary 6.2.6, the next result will be used to prove the converse of that corollary:

**Lemma 6.2.10.** Let $G$ be a circle graph, and let $\pi$ be a realizer for $G$ in which $S$ is consecutive. Assume that $G$ contains the split $(A, B)$ with frontiers $A'$ and $B'$, and let $S_a = S \cap A$ and $S_b = S \cap B$. Suppose that $S_a \neq A', \emptyset$ and $S_b \neq B', \emptyset$. Let $G_a$ and $G_b$ be the graphs resulting from the split operation of $G$ with respect to $(A, B)$, where $a \in V(G_a)$ and $b \in V(G_b)$ are the new vertices created by the operation. Assume further that $\pi$ is partitionable with respect to $(A, B)$. Then there is a realizer for $G_a$ in which $S_a \cup \{a\}$ is consecutive, and $a$ is a bookend; similarly, there is a realizer for $G_b$ in which $S_b \cup \{b\}$ is consecutive, and $b$ is a bookend.

**Proof.** We demonstrate the result for $G_a$ with the result for $G_b$ following symmetrically. Choose some $y \in B'$ and identify it with $a \in V(G_a)$. It is clear that $\pi[A \cup \{y\}] = \pi[A \cup \{a\}]$ is a realizer for $G_a$. We therefore focus on proving that $S_a \cup \{a\}$ is consecutive in $\pi[A \cup \{a\}]$, and that $a$ is a bookend.

Let $\gamma$ be the substring of $\pi$ certifying that $S$ is consecutive. Observe that since $\pi$ is partitionable, we can write $\pi = \alpha_1 \beta_1 \alpha_2 \beta_2$ such that the conditions of definition 6.2.7 are satisfied. The rest of the proof depends on the following claim:

**Claim:** No $\alpha_i$ or $\beta_i$ is a substring of $\gamma$. Suppose the opposite for contradiction. Without loss of generality, assume that $\alpha_1$ is a substring of $\gamma$. By remark 6.2.8, every vertex in $A'$ appears in $\gamma$; and by definition of consecutivity, no vertex in $A$ can appear more than once in $\gamma$. Therefore, since $S_a \neq A'$, there must be a substring $\delta$ of $\alpha_2$ that is also a substring of $\gamma$. This implies that $\beta_1$ is a substring of $\gamma$. Extending to $\beta_1$ the same reasoning applied to $\alpha_1$, we conclude that $\alpha_2$ is also a substring of $\gamma$, in addition to $\alpha_1$. Therefore, some vertex in $A' \subseteq A$ must appear more than once in $\gamma$, a contradiction.

From the above claim, we conclude that $S_a$ is consecutive in $\pi$. Let $\delta$ be the string certifying that $S_a$ is consecutive in $\pi$. Since $S_b \neq \emptyset$, we know that $\delta$ must either be a prefix or a suffix of $\alpha_1$ or $\alpha_2$. Of course, by remark 6.2.8, $y_1 \in \beta_1$ and $y_2 \in \beta_2$. Therefore $S_a \cup \{y\}$ is consecutive in $\pi[A \cup \{y\}]$, and $y$ is a bookend. The result follows by having identified $a$ with $y$. □

The result below is exactly the converse of corollary 6.2.6:

**Lemma 6.2.11.** Let $G$ be a circle graph and assume that $G + (x, S)$ is also a circle graph. Assume further that $G + (x, S)$ has a realizer in which $S$ is consecutive. If every node in $ST(G)$ is fully-mixed, then every one of its nodes satisfies the consecutive constraint with respect to $S$.  

126
Proof. Let $\pi_x$ be a realizer for $G + (x, S)$ in which $S$ is consecutive. Let $\pi = \pi_x[V(G)]$. Clearly $\pi$ is a realizer for $G$ in which $S$ is consecutive.

By remark 6.2.3 and the fact that $ST(G)$ is fully-mixed, it must happen that $ST(G)$’s nodes induce a path. Let $u$ be an endpoint on this path, and let $v$ be its only neighbour on this path. Let $q \in V(u)$ and $r \in V(v)$ be the markers of the edge $uv$. Then by remark 3.3.12, we know that $(L(q), L(r))$ is a split in $G$. As $ST(G)$ is fully-mixed, we have $S \cap L(q) \neq A(q)$ and $S \cap L(q) \neq \emptyset$. Furthermore, by lemma 6.2.9, we know that $\pi$ is partitionable with respect to $(L(q), L(r))$.

Let $G_a$ and $G_b$ be the graphs that result from the split operation with respect to $(L(q), L(r))$, where $a \in V(G_a)$, and $b \in V(G_b)$ are the new vertices that result from the operation. Choose some $y \in A(q)$. Let $G' = G'[V(G) - L(q)] \cup \{y\}$. Observe that $G_a$ is isomorphic to $G'$, with $y$ and $a$ mapped to each other. So by lemma 6.2.10, there is a realizer $\pi'$ for $G'$ in which $(S \cap L(r)) \cup \{y\}$ is consecutive, and $y$ is a bookend. Thus, $G(u)$ satisfies the consecutivity constraint, by its isomorphism with $G'$.

The case where $u$ is not an endpoint of this path is similar: merely apply the above argument to both of its neighbours on the path.

The precise conditions under which $G$ has a realizer in which $S$ is consecutive follow easily from corollary 6.2.6 and lemma 6.2.11. However, we are more interested in the resulting incremental split-tree characterization of circle graphs:

**Lemma 6.2.12.** Let $G$ be a circle graph such that every node in $ST(G)$ is fully-mixed, and consider some $S \subseteq V(G)$. Then $G + (x, S)$ is a circle graph if and only if every node in $ST(G)$ satisfies the consecutive constraint.

**Proof.** Necessity follows from lemma 6.2.11 and sufficiency from corollary 6.2.6.

In the next section, we combine the incremental characterization above with lemma 6.1.6, our earlier LBFS-incremental characterization of circle graphs. The result will form part of the foundation of our circle graph recognition algorithm.

### 6.3 Incremental Circle Graph Recognition

Let $G$ be a circle graph and consider the set $S \subseteq V(G)$. Assume that $x \notin V(G)$ is an end-vertex for $G + (x, S)$, and that every node in $ST(G)$ is fully-mixed. Then $G + (x, S)$ is a circle graph if and only if $G$ has a realizer in which $S$ is consecutive (lemma 6.1.6) if and only if every node in $ST(G)$ satisfies the consecutive constraint (lemma 6.2.12).
So consider some node $u$ in $ST(G)$. Assume that $u$ is degenerate. We have already observed that these graphs have chord diagrams of a special form (see figure 3.9). This fact allows us to easily determine if they satisfy the consecutivity constraint. It can be done according to the lemma below:

**Lemma 6.3.1.** Let $u$ be a degenerate node in a GLT. If $u$ is a clique, then $u$ satisfies the consecutive constraint if and only if $|M(u)| \neq 2$. If $u$ is a star with centre $c$, then $u$ satisfies the consecutive constraint if and only if $|M(u) \cup \{c\}| \neq 2$.

**Proof.** Sufficiency follows from algorithm 6.3.1 and lemma 6.3.2. For necessity, observe that if $u$ satisfies the consecutive constraint, then $|M(u)| \neq 2$, by remark 6.2.3. So consider a star node $u$ with centre $c$. If $\pi$ is a realizer for $G(u)$, then $\pi = \alpha c \beta$, where $\alpha$ and $\beta$ both certify that $V(u) - \{c\}$ are consecutive, and $\alpha = \cdots q \cdots r \cdots$ if and only if $\beta = \cdots q \cdots r \cdots$, for $q, r \in V(u) - \{c\}$. It is clear, then, that if $c \in M(u)$, then there is at most one $t \in V(u) - \{c\}$ such that $t \in M(u)$. □

Thus, algorithm 6.3.1 below suffices to determine if a degenerate node satisfies the consecutive constraint, and if so, constructs a realizer certifying this:

**Lemma 6.3.2.** Algorithm 6.3.1 is correct.

**Proof.** Straightforward from the definitions. □

We summarize the preceding discussion in the following lemma:

**Lemma 6.3.3.** Let $G$ be a circle graph and let $x \notin V(G)$ be a good vertex for $G + (x, S)$. Assume that every node in $ST(G)$ is fully-mixed with respect to $S$. Then $G + (x, S)$ is a circle graph if and only if for every node $u$ in $ST(G)$:

1. if $u$ is prime, then $u$ satisfies the consecutive constraint with respect to $S$;
2. if $u$ is a clique, then $|M(u)| \neq 2$;
3. if $u$ is a star with centre $c$, then $|M(u) \cup \{c\}| \neq 2$.

**Proof.** For necessity, assume that $G + (x, S)$ is a circle graph. Then $G$ has a realizer in which $N(x)$ is consecutive, by lemma 6.1.6. Thus, every node in $ST(G)$ satisfies the consecutive constraint, by lemma 6.2.12. Thus, condition 1 holds. Conditions 2 and 3 also hold, in this case by lemma 6.3.1.

For sufficiency, assume that conditions 1-3 hold for each node $u$ in $ST(G)$. By assumption, then, every prime node satisfies the consecutive constraint. Furthermore, if $u$ is degenerate, then it too satisfies the consecutive constraint, by lemma 6.3.1, with $NE(u)$ playing the role of $S$ and $M(u)$ playing the role of $S'$. Therefore, by lemma 6.2.12, $G + (x, S)$ is a circle graph. □
Algorithm 6.3.1: ConsecDegRealizer$(G, S, S')$

**Input:** A Degenerate Graph $G$ and non-empty sets $S', S \subseteq V(G)$ such that:

1. $S' \subseteq S$;
2. if $G$ is a star with centre $c$, then $|S' \cup \{c\}| \neq 2$;
3. if $G$ is a clique, then $|S'| \neq 2$.

**Output:** A realizer for $G$ in which $S$ is consecutive and each vertex in $S'$ is a bookend.

if $G$ is a star with centre $c$ then
   let $\alpha$ be a string that certifies that $S - \{c\}$ is consecutive, such that if $S' - \{c\} = \{q\}$, then $q$ is the last letter in $\alpha$;
   let $\beta$ be a string that certifies that $V(u) - (S - \{c\})$ is consecutive;
   return $ca\beta c\beta c^r \alpha^r$;
else
   let $\alpha$ be a string that certifies that $S$ is consecutive, with each vertex in $S'$ a bookend with respect to $\alpha$;
   return $\alpha\alpha$;
endif
Our circle graph recognition algorithm will implement the previous lemma. Determining if a prime node satisfies the consecutive constraint is a simple matter since it has a unique realizer, by lemma 3.4.5; given that realizer, we can use algorithm 6.3.2 for the task:

**Lemma 6.3.4.** Algorithm 6.3.2 is correct.

*Proof.* Straightforward from the definitions. □

**Algorithm 6.3.2:** ConsecPrimeRecog\((\pi, S, S')\)

*Input:* A realizer \(\pi\) for the prime graph \(G\), and non-empty sets \(S, S' \subseteq V(G)\) such that:

1. \(S' \subseteq S\);
2. \(|S'| \neq 2\).

*Output:* True if and only if \(S\) is consecutive in \(\pi\), and the vertices in \(S'\) are bookends.

pick some \(x \in S\);

assume that \(\pi = \cdots \alpha \cdots \beta \cdots\), where \(\alpha\) and \(\beta\) are the maximal substrings of \(\pi\) for which there are sets \(A, B \subseteq S\) such that \(x \in A, B\), and \(\alpha\) certifies that \(A\) is consecutive in \(\pi\) and \(\beta\) certifies that \(B\) is consecutive in \(\pi\);

assume that \(\alpha = a \cdots c\) and \(\beta = b \cdots d\);

if \(|\alpha| = |S|\) and \(S' \subseteq \{a, c\}\), or \(|\beta| = |S|\) and \(S' \subseteq \{b, d\}\) then return True;

else return False.

To apply algorithm 6.3.2, we will need to integrate our circle graph recognition algorithm with our LBFS-incremental split-tree algorithm. In particular, throughout the construction of the split-tree, we will maintain realizers at prime nodes. Integrating the two algorithms will also allow us to assume \(ST(G)\) is fully-mixed, as we did in lemma 6.3.3. The details of this integration follow next.

### 6.3.1 Algorithm

Our circle graph recognition algorithm is intended to be run in parallel with our LBFS-incremental split-tree algorithm. In what follows, we will assume that \(\sigma = x_1, \ldots, x_n\) is an LBFS of the graph \(G\), and that for each \(i\), \(G_i = G[x_1, \ldots, x_i]\). Assuming that \(G_{i-1}\) is a circle graph and that \(ST(G_{i-1})\) has been computed, our circle graph recognition algorithm will proceed to compute \(ST(G_i)\) according to algorithm 5.2.1, and then use that to determine if \(G_i\) continues to be a
circle graph. This is done by applying the next lemma, which merely translates to GLTs a result from [13, 68]:

**Lemma 6.3.5.** Let \( G = G(T, \mathcal{F}) \). Then \( G \) is a circle graph if and only if every label in \( \mathcal{F} \) is a circle graph.

**Proof.** By recursively performing node-joins in \((T, \mathcal{F})\), we arrive at a single node labelled by a graph isomorphic to \( G \). A realizer for this isomorphic graph can be obtained by inductively applying lemma 6.2.4, with the base case being our assumption that every label in \((T, \mathcal{F})\) is a circle graph and therefore has a realizer.

For necessity, assume that \( G \) is a circle graph. Let \( u \) be any node in \((T, \mathcal{F})\). To each \( q \in V(u) \), associate a leaf \( x \in A(q) \); let \( S \) be the set of leaves thus defined. Then clearly \( G[S] \) is isomorphic to \( G(u) \). So \( G(u) \) is a circle graph, by remark 3.4.2.

Now, degenerate graphs are always circle graphs, by remark 3.4.4. Hence, we merely need to determine if all prime nodes in \( ST(G_i) \) are circle graphs. However, all prime nodes in \( ST(G_i) \), except possibly the one adjacent to \( x_i \), are inherited from \( ST(G_{i-1}) \), by lemma 5.1.33. This reduces the problem to determining if \( x_i \) is adjacent to a prime node in \( ST(G_i) \), and if so, determining if that prime node is a circle graph. This task is facilitated by the correspondence summarized in the lemma below:

**Lemma 6.3.6.** Suppose that \( U \) is such that it consists of a single unmixed node from \( ST(G_{i-1}) \) if one exists, otherwise it consists of \( ST(G_{i-1}) \)'s fully-mixed nodes, of which there is more than one. Expand \( U \) by recursively node splitting each degenerate node \( u \in U \) according to \( (P^*(u), V(u) - P^*(u)) \) and \( (E^*(u), V(u) - E^*(u)) \) until no further node-split is possible. Let \( U' \) be the subset of \( U \) (after its expansion) consisting of \( U \)'s prime nodes and its non-fully-mixed degenerate nodes. Furthermore, assume that \( x_i \) is adjacent to a prime node \( u_x \) in \( ST(G_i) \), and that \( q \in V(u_x) \) is \( x_i \)'s opposite. Let \( G' = G(u_x) - q \). Then:

1. \( ST(G') \) contains a single node if and only if \( U \) consists of a single node; moreover, both of these nodes are prime and correspond to one another;

2. \( ST(G') \) contains more than one node if and only if \( U \) contains more than one node; moreover, there is a correspondence between the nodes in \( ST(G') \) and those in \( U' \).

**Proof.** A direct application of lemma 5.1.33.

To determine if the \( u_x \) in the lemma above is a circle graph, we apply lemma 6.3.3 to the \( ST(G') \). The entire process is summarized as algorithm 6.3.3:
Lemma 6.3.7. Algorithm 6.3.3 is correct.

Proof. By remark 3.4.2, $G$ is a circle graph if and only if $G_i$ is a circle graph, for every $i$. Therefore it suffices to show that if $G_{i-1}$ is a circle graph and the $i^{th}$ iteration of the loop in algorithm 6.3.3 is executed, then the algorithm will output $\text{False}$ if and only if $G_i$ is not a circle graph. To help the proof, we will further assume that $\mathcal{R}$ contains realizers for every prime node in $ST(G_{i-1})$.

If $G_{i-1}$ is a circle graph, then every node in $ST(G_{i-1})$ is a circle graph, by lemma 6.3.5. Of course, every degenerate graph is a circle graph, by remark 3.4.4. Moreover, by lemma 5.1.33, every prime node in $ST(G_i)$ is inherited from $ST(G_{i-1})$, except possibly the neighbour of $x_i$. Therefore, if $x_i$ is not adjacent to a prime node in $ST(G_i)$, then $G_i$ is a circle graph. Notice that algorithm 6.3.3 does not output $\text{False}$ in this case, and that $\mathcal{R}$ contains a realizer for every prime node in $ST(G_i)$.

So assume that $x_i$ is adjacent to a prime node $u_x$ in $ST(G_i)$. Then $G_i$ is a circle graph if and only if $G(u_x)$ is a circle graph. Let $q \in V(u_x)$ be the opposite of $x_i$. Let $G' = G(u_x) - q$. If $ST(G')$ contains a single node, then that node is vacuously fully-mixed. Otherwise, every node in $ST(G')$ must be fully-mixed, by lemma 6.3.6. Therefore, $G(u_x)$ is a circle graph if and only if every node in $ST(G')$ satisfies the conditions in lemma 6.3.3. Observe that algorithm 6.3.3 correctly verifies this and outputs $\text{False}$ precisely when they fail to hold, by lemmas 6.3.2 and 6.3.4. Moreover, if they do hold, then a realizer for $u_x$ is added to $\mathcal{R}$, by corollary 6.2.6. \hfill $\Box$

Algorithm 6.3.3 is the promised circle graph recognition algorithm. Its implementation and running-time are described below.

6.3.2 Implementation and Running Time

The implementation of algorithm 6.3.3 is simplified by its integration with algorithm 5.2.1. The latter computes the $\sigma$, $ST(G_i)$, and $ST(G')$ required by the former. It computes $\sigma$ and $ST(G_i)$ directly. For $ST(G')$, recall lemma 6.3.6 and note that its sets $U$ and $U'$ are computed indirectly by algorithm 5.2.1 through algorithm 5.1.5.

We will use the implementation of algorithm 5.2.1 described in section 5.3. Hence, we can further assume that $P(u)$ and $M(u)$ have been indirectly computed for each node $u$ in $ST(G')$, by remarks 5.3.7 and 5.3.8. The implementation for the remainder depends on the data-structure we adopt for realizers, described below.

Data Structure: Partially Complemented Realizers

Let $G$ be a circle graph and let $\pi$ be a realizer for $G$. We will represent $\pi$ by a collection of data-objects, one for each letter in $\pi$, with each data-object consisting of two pointers: $+$ and $-$. 132
Algorithm 6.3.3: CircleGraphRecog(G)

Input: A graph G.

Output: A realizer for G if and only if G is a circle graph, otherwise False.

\( \sigma \leftarrow x_1, \ldots, x_n \), an LBFS of G;
\( \mathcal{R} \leftarrow \emptyset \);

\textbf{foreach } \( i = 1 \text{ up to } n \) \textbf{do}

Form ST\((G_i)\) from ST\((G_{i-1})\) using AddVertex\((ST(G_{i-1}), x_i, N(x_i) \cap V(G_i))\);

// algorithm 5.1.5

\textbf{if } x_i \text{ is adjacent to a prime node } u_x \text{ in } ST(G_i) \text{ then}

\hspace{1em} let \( q \in V(u_x) \) be the opposite of \( x_i \), and let \( G' = G(u_x) - q \);
\hspace{1em} assume marker vertices in ST\((G')\) are assigned states with respect to \( N(x_i) \cap V(G_i)\);
\hspace{1em} \( \mathcal{R}' \leftarrow \emptyset \);

\textbf{foreach node } u \text{ in } ST(G') \text{ do}

\hspace{2em} \textbf{if } u \text{ is a star with centre } c \text{ and } |M(u) \cup \{c\}| \neq 2 \text{ then}
\hspace{3em} \mathcal{R}' \leftarrow \mathcal{R}' \cup \{\text{ConsecDegRealizer}(G(u), NE(u), M(u))\};
\hspace{2em} // algorithm 6.3.1

\hspace{2em} \textbf{else if } u \text{ is a clique and } |M(u)| \neq 2 \text{ then}
\hspace{3em} \mathcal{R}' \leftarrow \mathcal{R}' \cup \{\text{ConsecDegRealizer}(G(u), NE(u), M(u))\};
\hspace{2em} // algorithm 6.3.1

\hspace{2em} \textbf{else if } u \text{ is prime, } \pi \in \mathcal{R} \text{ is a realizer for } G(u), \text{ and}
\hspace{3em} \text{ConsecPrimeRecog}(\pi, NE(u), M(u)) \text{ // algorithm 6.3.2}
\hspace{2em} \text{ then}
\hspace{4em} \mathcal{R}' \leftarrow \mathcal{R}' \cup \{\pi\}
\hspace{2em} \text{ else return False}

\textbf{endif}

\textbf{endif}

\( \mathcal{R} \leftarrow (\mathcal{R} \cup \text{ConsecContractAdd}(ST(G'), \mathcal{R}', q)) - \mathcal{R}' \);

\textbf{endfch}

return \( \mathcal{R} \);
If $x \in V(G)$, then $\pi = \cdots ax_1b \cdots cx_2d \cdots$, and the data-objects for $x_1$ and $x_2$ will satisfy the following constraints:

- **Completeness Constraint**: $(+x_1, -(x_1)) = \{a, b\}$ and $(+x_2, -(x_2)) = \{c, d\}$;
- **Consistency Constraint**: $+(x_1) = b$ if and only if $+(x_2) = d$.

It needs to be emphasized that the $+/-$ pointers do not correspond to the predecessor/successor relation in a realizer. An example for the chord diagram in figure 2.3 could be:

- $+(e_1) = f_2, -(e_1) = b_1, +(e_2) = d_1, -(e_2) = c_2$;
- $+(b_1) = c_1, -(b_1) = e_1, +(b_2) = a_2, -(b_2) = a_1$;
- $+(c_1) = a_1, -(c_1) = b_1, +(c_2) = d_2, -(c_2) = e_2$;
- $+(a_1) = c_1, -(a_1) = b_2, +(a_2) = b_2, -(a_2) = f_1$;
- $+(f_1) = d_1, -(f_1) = a_2, +(f_2) = e_1, -(f_2) = d_2$;
- $+(d_1) = f_1, -(d_1) = e_2, +(d_2) = c_2, -(d_2) = f_2$.

Notice that $+(e_1)$ refers to $e_1$’s counter-clockwise “neighbour” while $+(b_1)$ refers to $b_1$’s clockwise “neighbour”. In this way, the “neighbourhoods” of $e_1$ and $b_1$ are “complemented” with respect to each other. Hence the name we give to this data-structure: partially-complemented realizer (PCR).

This type of complementation is responsible for the following observation:

**Remark 6.3.8.** A PCR for $\pi$ is simultaneously a PCR for any rotation of $\pi$, and any reversal of these.

The preceding remark was the intent of designing PCRs. Their invariance under rotation and reversal will be needed for the efficient implementation of algorithm 6.2.1. Of course, prior to that algorithm being called by algorithm 6.3.3, it must first be verified that $G_i$ is a circle graph.

**Verifying Degenerate Consecutivity**

Recall $ST(G')$ from algorithm 6.3.3, and the correspondence between its nodes and the set of $ST(G_{i-1})$’s (unmixed/fully-mixed) node(s) as per lemma 6.3.6. As explained earlier, we can assume that $P(u)$ and $M(u)$ (and thus $NE(u)$) have been computed. Of course, the data-structure adopted for $ST(G_{i-1})$ provides constant time access to the centre of every star node. Hence, for
each degenerate node $u$ in $ST(G')$, it can be determined in constant time if it satisfies the consecutive constraint, as in the tests in the first two branches of the conditional in the nested loop of algorithm 6.3.3.

If every such degenerate node satisfies the consecutive constraint, then algorithm 6.3.1 is called to create realizers certifying this. The cost of creating each realizer is clearly proportional to the size of the degenerate node in question. Thus, by corollary 5.4.13:

**Lemma 6.3.9.** The cost associated with algorithm 6.3.1 in the running-time of algorithm 6.3.3 is $O(n + m)$.

Just as importantly, algorithm 6.3.1 allows us to conclude the following of the realizers it produces:

**Remark 6.3.10.** In the implementation of algorithm 6.3.3, we can assume that substrings have been identified certifying that each degenerate node satisfies the consecutive constraint.

**Verifying Prime Consecutivity**

Unlike for degenerate graphs, realizers for prime nodes in $ST(G_{i-1})$ are not created; instead, algorithm 6.3.3 assumes they exist in the set $\mathcal{R}$. We will correspondingly extend the implementation of algorithm 5.2.1 presented in section 5.3 by maintaining, for each prime node $u$, a PCR for its label $G(u)$. These are intended to correspond to $\mathcal{R}$. Each marker vertex $q \in V(u)$ will maintain pointers to the data-objects corresponding to its two instances in the PCR. This does not effect the implementation nor the running-time of algorithm 5.2.1, which operates independently of this additional information. We take the existence of these PCRs for granted now and afterwards show how they are maintained.

To determine if a prime node satisfies the consecutive constraint, algorithm 6.3.3 calls algorithm 6.3.2. The key in that latter algorithm is the computation of the substrings $\alpha$ and $\beta$. Once these have been computed, the rest of the algorithm becomes constant time. So assume that $\pi = \cdot \cdot \cdot abx_1cd\cdot \cdot \cdot$. Then $\alpha$ can be computed by starting at $x_1$, and first considering $b$. If $b \in S$, then we go on to process $a$; if $a \in S$, then we continue on again to the letter preceding $a$, otherwise we stop. We do the same in the direction of $c$, and so on. Proceeding similarly for $x_2$, we compute $\beta$.

In terms of PCRs, $a$ and $b$ can be identified by $+(x_1)$ and $-(x_1)$, given the completeness constraint. Importantly, we do not care which of $+(x_1)$ and $-(x_1)$ is $a$ or $b$, just that they identify them. Say $-(x_1) = b$. To identify $a$, we check which of $+(b)$ and $-(b)$ is not $x_1$. And so on for
the letter preceding $a$. Similarly for the direction of $c$, and for $x_2$. Hence, the running-time of algorithm 6.3.2 is $O(|S|)$.

Now, algorithm 6.3.3 calls algorithm 6.3.2 for each prime node $u$ in $ST(G')$. The inputs are $\pi, NE(u)$, and $M(u)$, where $\pi \in \mathcal{R}$ is a realizer for $G(u)$. As already explained, we can assume that $P(u)$ and $M(u)$ (and thus $NE(u)$) have been computed. The cost of each call to algorithm 6.3.2 is therefore $O(|NE(u)|)$, which is $O(|V(u)|)$. Recall the correspondence of $ST(G')$ to $ST(G_{i-1})$’s (unmixed/fully-mixed) node(s) guaranteed by lemma 6.3.6. Thus, by lemmas 5.1.28 and 5.4.14:

**Lemma 6.3.11.** The cost associated with algorithm 6.3.2 in the running-time of algorithm 6.3.3 is $O(n + m)$.

We also clearly have the following remark, similar to the one for degenerate realizers:

**Remark 6.3.12.** In the implementation of algorithm 6.3.3, we can assume that substrings have been identified certifying that each prime node satisfies the consecutive constraint.

**Constructing Prime Realizers**

Each prime node not adjacent to $x_i$ is inherited from $ST(G_{i-1})$, by lemma 5.1.33. Hence, realizers for these nodes will persist into $ST(G_i)$. If $x_i$ is adjacent to a prime node, then a realizer for that node is formed according to algorithm 6.2.1 (see algorithm 6.3.3).

Consider the loop in algorithm 6.2.1, and the nodes $u$ and $u'$ and marker vertices $q$ and $r$ there defined. By lemma 6.3.6, the node-join described during each iteration corresponds to a node-join effected by algorithm 5.2.1. We can therefore assume that we have access to the markers $q$ and $r$. From these, access to their instances in the appropriate PCRs can be obtained in constant time by following the pointers for this purpose.

With this, PCRs for each of the eight realizers listed in the loop can be formed in constant-time: given the consistency constraint, the appropriate substrings can be identified in constant time; since PCRs are invariant under rotation and reversal, nothing is required for the reversal of these substrings; finally, only a constant number of pointer updates are required to stitch the substrings together.

The input to algorithm 6.2.1 is a split-tree $ST(G)$, each of whose nodes is fully-mixed. Therefore $q$ and $r$ are mixed. Furthermore, $ST(G)$’s nodes all satisfy the consecutive constraint. Thus, any string certifying the consecutive constraint for $u$ is a prefix or suffix of $\alpha$ or $\beta$; similarly, any string certifying the consecutive constraint for $u'$ is a prefix or suffix of $\gamma$ or $\delta$. By remarks 6.3.10 and 6.3.12, we can assume certifying strings have been identified. It is therefore a constant time
operation to determine which PCR corresponds to the realizer \( \pi \) in algorithm 6.2.1: as pointers are updated to form each PCR, check to see if the two certifying strings are concatenated.

Combining our observations, we conclude that the cost of each iteration of the loop in algorithm 6.2.1 is constant. The number of iterations is proportional to the number of nodes in the split-tree input to the algorithm. We are interested in calls to algorithm 6.2.1 made by algorithm 6.3.3. Examining the latter, we see that the input graph is \( ST(G') \), whose nodes correspond to \( ST(G_{i-1}) \)'s (unmixed/fully-mixed) node(s), by lemma 6.3.6. So by lemmas 5.1.28 and 5.4.14:

**Lemma 6.3.13.** The cost associated with algorithm 6.2.1 in the running-time of algorithm 6.3.3 is \( O(n + m) \).

**Running-Time**

By combining lemmas 6.3.9, 6.3.11, and 6.3.13, we see that the portion of algorithm 6.3.3 not relying on algorithm 5.2.1 runs in time \( O(n + m) \). Of course, by theorem 5.4.15, algorithm 5.2.1 runs in time \( O(\alpha(n + m) \cdot (n + m) + n + m) \), where \( \alpha \) is the inverse of Ackermann’s function. Therefore:

**Theorem 6.3.14.** Circle graphs can be recognized in time \( O(\alpha(n + m) \cdot (n + m) + n + m) \), where \( \alpha \) is the inverse of Ackermann’s function.

This is an improvement on the previous \( O(n^2) \) bound for circle graph recognition. In deriving the bound, we developed a previously unknown LBFS-characterization of circle graphs by introducing the notion of consecutivity (lemma 6.1.6). This was extended in deriving a split-tree characterization of circle graphs (lemma 6.2.12). We also translated an existing split decomposition characterization of circle graphs to the GLT setting (lemma 6.3.5). These three characterizations combined to suggest algorithm 6.3.3, which proves the bound.

Of course, the bound depends on algorithm 6.3.3 being integrated with algorithm 5.2.1. This further argues for the latter as it relates to other split decomposition algorithms. Yet, we see above that the algorithm also proves to be the only barrier to linear-time circle graph recognition. This impediment is discussed in detail in the conclusion of this thesis.
Chapter 7

Conclusion

Graph decomposition has had a profound impact on the study of graph theory. Hierarchical decompositions, in particular, have been especially important in the design of graph algorithms. Two of the most important examples in this regard are modular decomposition and split decomposition. The fundamental problem in algorithmic applications of hierarchical decompositions is the computation of their decomposition tree. This thesis presented improved algorithms for computing the modular decomposition tree and and split decomposition tree (chapters 4 and 5, respectively).

Both our algorithms are simpler than earlier solutions. The modular decomposition algorithm is optimal in that it runs in linear-time. The split decomposition algorithm requires union-find, meaning its running-time includes an inverse of Ackermann factor. All other parts of the algorithm, however, are consistent with linear-time, and the inverse of Ackermann grows so slowly as to be effectively constant (pg. 521-522, [33]). The primary advantage of our split decomposition algorithm over linear-time alternatives is its ability to extend to circle graph recognition. We are also the first to employ GLTs in computing split decomposition.

A common theme in all three algorithms is the application of LBFS. In each case it has the effect of simplifying the algorithm in question. We revisit each algorithm below to address outstanding questions and suggest directions for future research.

7.1 Modular Decomposition

The modular decomposition algorithm presented in this thesis is conceptually simple, and practical in its implementation. Its correctness only relied on the basic notions of a module, its type, and its manifestation in the MD tree (lemma 3.2.4). Its implementation only required an ordered list of trees and simple traversals of these trees. The algorithm unifies three streams of concurrent
research that all had as their goal such an algorithm. Prior to that, there had been a long succession of modular decomposition algorithms offering incremental advancements. We hope that our algorithm represents the natural conclusion of this area of research – that it will become the modular decomposition algorithm, and the choice for any practical application.

7.1.1 Practical Applications

The case has been made for a simple and practical modular decomposition algorithm by appealing to its many algorithmic applications. A recent commentary notes the difficulty of actually implementing earlier modular decomposition algorithms [120]. The authors of [70] analyze protein-protein interaction networks using the modular decomposition algorithm of [103], and qualify their choice by saying it has a practical implementation (pg. 11,[70]). The drawback with that algorithm is that it runs in time $O(n + m \log n)$. Partly to remedy this, but mostly to prove its practical implementation, we coded a preliminary version of our modular decomposition algorithm (see [130] for this version) and made it publicly available [128].

We have since heard from three different researchers using our code. William Wu at Columbia University used our code to heuristically gauge the perfection of an experimental database of graphs arising from his work in discrete harmonic analysis [134]. A more traditional example is provided by a group at Texas A&M University, who obtained an improved algorithm for the cluster editing problem by using modular decomposition. They used our code to implement their algorithm and perform experiments with it [21]. Another typical example is Heinis’ work modelling scientific workflows as a service [88]. Our modular decomposition code is used as a subroutine in a transitive orientation algorithm that recognizes interval containment graphs. Such graphs can be encoded and stored in a database to facilitate reachability queries that answer questions on data provenance and data linkage [89]. He notes that this recognition phase empirically proved the most time-consuming, and that switching to our linear-time implementation significantly improved performance (pg.128, [88]).

7.1.2 Transitive Orientation

The transitive orientation problem is to determine if the edges of an undirected graph can be oriented in such a way that the resulting directed graph is transitive. Graphs for which this is possible are called comparability graphs. These have been intensely studied in their own right, as well as in relation to their numerous applications, notably scheduling theory (see e.g. [104]), of

---

1The finalized version of the algorithm seems to have moved away from modular decomposition, see [22]
which Heinis’ work above is suggestive. The books [82] and [16] provide an overview of this class of graphs.

As mentioned in chapter 2, modular decomposition arose in part from the investigation of transitive orientation and comparability graphs [71]. The connection is analogous to that observed in chapter 2 between circle graph recognition and split decomposition: the problem can be reduced to graphs that are prime (with respect to modular decomposition). This is the approach used by all efficient transitive orientation algorithms (see e.g. [81, 101, 102]). Prime comparability graphs have the nice property of having a unique transitive orientation\(^2\). Spinrad pioneered an elegant partition refinement strategy for computing this unique orientation, based on the idea of pivoting [121, 122]. We borrowed the notion of a pivot in our modular decomposition algorithm.

There is only one linear-time transitive orientation algorithm [102]. It begins by applying the linear-time modular decomposition algorithm first described in [101] to reduce the problem as above. Spinrad’s partitioning algorithm is then applied. Information gathered from the modular decomposition computation is used to determine an optimal pivot sequence. This sequence is what achieves linear-time.

Spinrad’s original work described a clever pivot selection rule that produces an \(O(n + m \log n)\) implementation. Although not as fast, it is a beautiful algorithm. The linear-time alternative is not. It interacts with and further complicates a modular decomposition algorithm that was already deemed impractical and of only theoretical interest (see [56], pg. 2, for example).

The natural question is whether our modular decomposition algorithm can be extended to solve the transitive orientation problem simply and in linear-time. The answer is yes. Like [102], we do not just use modular decomposition to reduce the problem. In our case, we add an extra partitioning procedure at each recursive step. This allows us to determine the relative direction of the active edges at that step. By “relative”, we mean that once the direction of one edge is determined, the direction of all others are implied. Thus, we need only find the direction of a single edge, and then follow the chain of implications. Spinrad’s elegant partitioning algorithm can do this. We are currently preparing our algorithm for journal submission.

7.1.3 Comparative Analysis

The study of modular decomposition itself is by now fairly mature. Our modular decomposition algorithm reflects this in its unifying approach. Even if our algorithm becomes the modular decomposition algorithm, the subject will remain interesting for its applications. Transitive orientation

\(^2\)Up to reversing the directions on all edges.
is a classical example, and Heinis’ work mentioned above demonstrates its continued relevance. More novel applications are the harmonic analysis and bioinformatics settings also mentioned. The limitations/boundaries/questions/solutions arising from these multi-disciplinary applications need to be explored and reflected in the graph literature. We see the potential already in examples from chapter 2. There, we alluded to two algorithms from computational biology that suggested algorithms to compute the MD tree from a factorizing permutation [136, 5].

Insight must also be incorporated from other decomposition schemes. Modular decomposition and related decomposition schemes have suffered from being studied in isolation. Bui-Xuan and Habib have developed a common generalization of various decomposition schemes, including modular decomposition, rank-width, and branch-width [135]. The existence of such generalizations suggest research potential in the application of ideas from one to the other. More generally, a comparative analysis between decomposition schemes is lacking. Why are problems amenable to one, not amenable to another? On what graphs? Can we characterize differences in terms of structure revealed by different decomposition schemes? What are the “boundaries” in this regard? These are all broad questions that need to be defined.

One context for their definition may be provided in [137]. They define a framework for viewing modular decomposition, split decomposition, and rank-width as instances of a generic $H$-join decomposition based on bipartite graphs. By comparatively studying the different bipartite graphs they associate with each decomposition scheme, we may get a better understanding of the relationship between these schemes. We believe that further progress in the theory of modular decomposition can only come by expanding the context of its investigation.

### 7.2 Split Decomposition

There had been a need for a linear-time split decomposition algorithm that could be understood, verified, and implemented. The only linear-time split decomposition algorithm was too complicated in all these aspects. We presented a split decomposition algorithm in this thesis that succeeds in most senses. It is conceptually simple and easily verified; its implementation only requires rooted trees and union-find; and it is efficient, only sacrificing linear-time for its use of union-find. The only difficulty in the algorithm is the proof of its running-time, which is independent of the algorithm itself.
7.2.1 Linear-Time and Union-Find

The bottleneck in our split decomposition algorithm is the cost of updating parent pointers after a node-join. This would not be the case if every root marker vertex in the split-tree was universal in its label. Then, every marker vertex of a node would accept a new incident edge as a result of the node-join. Thus, the cost of updating the parent pointer could be charged to the cost of creating this new label-edge. Union-find would not be needed. Overall, the cost would be linear-time, by corollary 5.4.13.

There are special cases of the split-tree where every root marker vertex is universal in its label. It was shown in [78] that the MD tree can be represented by GLTs with this property. Thus, as a corollary of our split-decomposition algorithm, we get a linear-time modular decomposition algorithm. It is debatable whether such an algorithm is simpler than the modular decomposition algorithm presented in chapter 4. Certainly the running-time proof of the split-tree version is nowhere near as simple.

Gabow and Tarjan demonstrate a linear-time implementation of union-find for the special case where the unions are known in advance [69]. A “union-tree” must be computed a priori such that each node corresponds to a union in the sense that its descendant leaves correspond to the elements in the resulting set. We believe that such a tree can be defined in our case based on the slice structure of the LBFS ordering in which vertices are inserted. However, our claim to a simpler algorithm would be sacrificed this way. Gabow and Tarjan’s implementation is not simple. It requires advanced, low-level programming tricks, operating at the bit-level.

7.2.2 Linear-time Without Union-Find

We saw earlier in figure 5.8 that the worst-case time-complexity of adding a new vertex to the split-tree is $O(n)$. In the example there, the problem was the creation of a large induced cycle, which caused the split-tree to collapse from $O(n)$ nodes into one. If we nest such cycles as in figure 7.1, then we will observe repeated contractions. Unsurprisingly, this is very costly. In fact, the example in figure 7.1 provides an $\Omega(n^{1.5})$ lower-bound for the implementation of our algorithm without union-find – where parent pointers are updated in the natural way. An entirely different implementation is therefore required for our algorithm to achieve linear-time.

Or it may be that an entirely different approach is necessary to achieve linear-time. We have developed an approach to computing the split-tree that follows the recursive LBFS structure employed in our modular decomposition algorithm. As in that algorithm, the recursive subproblems would be defined by maximal LBFS slices. In this case, however, the split-tree would be recursively
Figure 7.1: An input graph that provides a lower-bound for the implementation of our split decomposition algorithm \textit{without} union-find. The required LBFS ordering is produced by starting at the root, and exploring the tree by always choosing the leftmost unexplored child. Contraction would result from processing each vertex on the bottom layer of the tree. The resulting number of parent pointer updates would be $\Omega(n^{1.5})$.

computed. Each of the recursively computed split-trees will be subtrees of the split-tree for the entire graph, although possibly after some (or all) parts of them have been contracted.

What parts need to be contracted is determined by a type of refinement. If $P_i$ and $P_j$ are two maximal slices, $i < j$, and vertices in $P_i$ are universal to $P_j$, then vertices in $P_j$ must be used to refine the split-tree for $P_i$, as was done for modular decomposition. The refinement in this case involves adding a single “representative” from $P_j$ to the split-tree for $P_i$. If this is the only set of adjacencies to vertices in $P_j$, then the split-tree for $P_j$ can simply be grafted to the split-tree for $P_i$ via this “representative”.

If there is also a $P_{\ell}, \ell < i$, that has vertices universal to $P_j$, then a “representative” must also be added to the split-tree for $P_{\ell}$. The path from the representative in $P_i$ to the root must be collapsed into a single vertex. Then $P_i$, $P_j$, and $P_{\ell}$ can all be grafted together via the representatives. Otherwise, there may be another slice to which $P_j$ has adjacencies, and the pattern would be the same, only now the path from the representative to the root of the split-tree for $P_{\ell}$ would have to be collapsed as well. A type of closure operation can be used to define what must collapsed, and how all the recursively computed trees are grafted into one.

Like modular decomposition, the running-time would be characterized in terms of active edges between maximal slices. The addition of each representative would follow the incremental algorithm presented in this thesis. We could use the fact that vertices are either universal to or isolated from maximal slices in order to avoid using union-find. However, the charging argument would still be needed. It would therefore be an extension of the algorithm presented in chapter 5. We feel the current description is long enough as it is. We have not formalized our idea for this reason.
7.2.3 Simpler Split-Decomposition?

The impracticality attributed to the linear-time modular decomposition algorithm in [56] is largely a result of it using the Gabow-Tarjan version of union-find. The same paper, however, does present a more practical $O(m\alpha(n, m))$ alternative using the traditional union-find implementation; as always, $\alpha$ is the inverse of Ackermann’s function. In many ways, our split decomposition algorithm is its analogue. Both papers use union-find to simulate parent pointers and require a careful and detailed charging argument to prove their running-time. Hence, the natural question to ask is if there is a simpler, linear-time split decomposition algorithm in the way that we demonstrated a simpler, linear-time modular decomposition algorithm in chapter 4.

We are interested in a split decomposition algorithm that is composed of basic operations on elementary data-structures, that is transparent in its correctness and running-time. One place to start would be the reimagining of Dahlhaus’ algorithm in [27]. Much of their framework is already implicit in the structure of GLTs. For example, their *strong splits* are precisely those encoded by the split-tree’s tree-edges (recall as well, theorem 3.3.17). It would be interesting to see the result of applying Dahlhaus’ ideas to GLTs.

As it is, the restated algorithm in [27] is still fairly advanced. It requires a modular decomposition as a subroutine, as well as an algorithm of McConnell from [99], itself adapted from Dahlhaus’ original algorithm. Unfortunately, neither are specified in the paper, leaving that difficulty to the reader. A self-contained presentation would be preferred.

7.3 Circle Graphs

The quadratic bound on circle graph recognition had stood for fifteen years. This thesis derived a circle graph recognition algorithm that improved this bound. The algorithm is based on a new characterization of circle graphs in terms of LBFS. As explained in chapter 1, we noticed this relation through properties revealed by the split-tree. Other properties of the split-tree allow us to define induced LBFS orderings at each node. We can therefore carry our characterization through to each node without having to explicitly compute an LBFS for each label. This helps guarantee the algorithm’s efficiency. The other component in its efficiency is a new reflection-invariant data-structure we develop for chord diagrams.
7.3.1 Linear-Time?

A linear-time implementation of our split-tree algorithm would immediately imply linear-time circle graph recognition. This provides some incentive for investigating the possibility of applying the Gabow-Tarjan version of union-find to our split decomposition algorithm – despite its impracticality. Unlike split-decomposition, there is no existing linear-time circle graph recognition algorithm; it would be an interesting theoretical result to prove a linear-time upper bound. Given the discussion surrounding figure 7.1, we are pessimistic about the possibility of removing union-find altogether. Thus, given the lower-bounds that exist for union-find [116], a linear-time implementation seems unlikely without Gabow-Tarjan.

Nevertheless, we did allude above to the potential for a linear-time split decomposition algorithm based on the recursive LBFS approach adopted for modular decomposition. We are also confident that our LBFS characterization of circle graphs could be adapted to this recursive setting. The result would be a linear-time circle graph recognition algorithm. Unfortunately, the same problems cited above would remain. Namely, that it would require our incremental algorithm as a subroutine, along with its charging argument, thereby making an already long paper even longer. This is also the reason we did not formalize the possible circle graph extension.

7.3.2 Linear-Time Recognition of Permutation Graphs

*Permutation Graphs* are the intersection graphs of chords whose endpoints lie on opposite parallel lines. Equivalently, they are the circle graphs whose chord diagrams permit the addition of a single chord intersecting all others. Another way of representing them is as two permutations of the set \( S = \{1, \ldots, n\} \), such that each \( i \in S \) is a vertex, and vertices \( i \) and \( j \) are adjacent if and only if their order is reversed in the two permutations. In each case, the relationship to circle graphs is clear (see figure 7.2). In fact, there exists an LBFS characterization of permutation graphs similar to the one we developed for circle graphs; we omit it in the interests of space.

The same approach recognizing circle graphs can be applied to recognize permutation graphs. The only difference is that although realizers for prime permutation graphs are unique, we mean prime in the sense of modular decomposition, not split decomposition. Hence, we maintain the GLT version of the MD tree, which is outlined in [78]. As we have already noted, every root marker vertex in this tree is universal in its label. Thus, the GLT version of the MD tree can be constructed in linear-time, for the reasons mentioned above. The result is a linear-time permutation graph recognition algorithm.

Such an algorithm would be interesting for being the first to recognize permutation graphs
Figure 7.2: A permutation graph on the right, and its two different interpretations on the left. The dotted lines demonstrate how the intersection model can be transformed into a chord diagram, explicitly demonstrating the connection to circle graphs.

“directly”. It is known that permutation graphs are precisely the comparability graphs whose complement is also a comparability graph. Using this fact, the other linear-time recognition algorithms attempt to compute transitive orientations of the input graph and its complement, and then check if they were successful in doing so. If yes, then the graph is a permutation graph, otherwise it is not [101, 102]. Our algorithm would be the first to avoid transitive orientation. It could instead be viewed as a special case of the fully-dynamic permutation graph recognition algorithm of [49]: instead of adding vertices arbitrarily, vertices are added according to an LBFS. We are currently preparing this result for submission.

7.3.3 Graph-Labelled Trees and their Generalizations

Our split-decomposition algorithm has the advantage over [27] of extending to circle graph recognition. We owe this fact to the split-tree. We have already mentioned its role in discovering the new LBFS characterization of circle graphs. What else it reveals about circle graphs is still to be explored. A thorough investigation of its properties in terms of distance-hereditary graphs was undertaken in [78]. The same needs to be done for circle graphs and other graph classes that have proved amenable to split decomposition; for example, parity graphs [32] and circular arc graphs [92].

The split-tree’s potential in traditional applications of split decomposition is still unknown. What does it reveal about clique-width or rank-width? Does its structure help us say anything about these two decompositions? Does the split-tree help to provide a better graph-theoretic interpretation of rank-width, as has recently been attempted in [45, 137]? One could foresee structural results including characterizations based on configurations or properties of the split-tree. Since the split-tree generalizes both the MD tree and the split decomposition tree [78], what role can it play in the type of comparative analysis we described earlier? Since most graphs are prime,
are generalizations of the split-tree needed to have an impact in this way? We believe that GLTs are a rich and flexible combinatorial object that can be adapted to answer some of these questions.

7.4 Lexicographic-Breadth First Search

LBFS played a role in the each of the three main results presented in this thesis. In general, it is frequently successful in the recognition of classes of graphs that are defined by vertex orderings. Quite often these recognition algorithms employ sophisticated tie-breaking schemes. The modular decomposition algorithm presented in this thesis is the first to apply a retroactive tie-breaking scheme. Later, we used LBFS to define circle graphs in terms of their LBFS orderings, and then translated this characterization to the split-tree for its efficient implementation. There, LBFS was used to streamline the effects of adding each new vertex during the split-tree’s incremental construction. A similar effect on PQ-trees had famously been observed by Korte and Mohring [94]. We are the first to establish connections between LBFS and circle graphs, and LBFS and split decomposition.

7.4.1 A New LBFS Paradigm

The modular decomposition algorithm presented in this thesis formulated a new recursive approach for computing an LBFS. The recursion produces an ordered list of MD trees. These are then refined to produce a factorizing permutation. We can view this refinement as a kind of retroactive tie-breaking. In fact, this recurse-and-refine approach defines a new LBFS paradigm.

In place of the MD tree could be any combinatorial object, and the type of refinement could be adapted accordingly. LBFS works “forward” to help define the recursive subproblems, and then a “backward” refinement occurs, the details of which would depend on the application. Ultimately, the result is an ordering of the vertices. Another way of looking at it is that LBFS defines a rough outline for the ordering, and the backward refinement defines the ordering within this LBFS outline. The final output may no longer be an LBFS ordering, but the point is not so much that, as it is to take advantage of the structural properties of the graph revealed by LBFS throughout its computation. Below we discuss another possible application.

7.4.2 Recognition of Chordal Bipartite Graphs

Chordal Bipartite Graphs generalize the concept of chordal graphs. They are the bipartite graphs having no induced cycle of length greater than 4. An equivalent characterization is with respect to their adjacency matrix. A graph is chordal bipartite if and only if every doubly-lexical ordering
of its adjacency matrix does not contain $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$ as a submatrix – the so-called $\Gamma$ [66, 97]. A doubly-lexical ordering of an adjacency matrix is an ordering of its rows and columns such that when the rows and columns are viewed as vectors, they are seen to be monotonically increasing in value. The fastest chordal bipartite recognition algorithms all operate by first computing a doubly-lexical ordering, and then verifying that it is $\Gamma$-free. The verification can be done in linear-time; the bottleneck is the doubly-lexical ordering, which takes time $O(\min(m \log n, n^2))$ [112, 125].

Tantalizingly, every lexical ordering can be realized by LBFS. In other words, there is a way of breaking ties in an LBFS to generate any doubly-lexical ordering. This is not surprising: chordal bipartite graphs can be characterized by vertex orderings; moreover, they do not contain large induced cycles. LBFS has been especially successful on classes of graphs sharing these two properties. The problem is: not every LBFS ordering produces a lexical ordering. The question therefore becomes how to break ties.

The problem is hard for the following reason. Let $G$ be a chordal bipartite graph, and let $A$ and $B$ be its colour classes. Then $G + x$ is also a chordal bipartite graph, where $N(x) = A$. If we start an LBFS at $x$, then the vertices in $A$ will all be tied. Because there are no edges between them, we must essentially choose them at random for exploration. This easily leads to making the wrong choice when our goal is a lexical ordering. The problem propagates to induced subgraphs. Based on this fact, we have been able to construct examples showing that traditional, multi-sweep LBFS tie-breaking schemes (see chapter 2 and [34]) cannot work.

Ideally, when we make a bad choice in the situation described above, we would like the option of correcting it later. This suggests the backward refinement paradigm just described. However, recursing within slices as we do for modular decomposition will not work because each slice in a bipartite graph is an independent set. One idea is to recurse within the graph induced by the slice and its neighbourhood – vertices adjacent to those in the slice. We have developed an algorithm based on this idea in hopes of recognizing chordal bipartite graphs in linear-time. Initial results have been promising.

A linear-time chordal bipartite graph recognition algorithm based on LBFS was claimed in [132], but this paper was later found to have an error noted at [131]. A correct solution would imply a host of other results, including linear-time algorithms for many combinatorial optimization problems on chordal bipartite graphs. The linear-time recognition of chordal bipartite graphs remains one of the big open problems in algorithmic graph theory.
Bibliography


158


## Index

<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A(q)$, 25</td>
<td></td>
</tr>
<tr>
<td>Charge$_d$, 105</td>
<td></td>
</tr>
<tr>
<td>Charge$_e$, 105</td>
<td></td>
</tr>
<tr>
<td>$D_c$, 103</td>
<td></td>
</tr>
<tr>
<td>$D_v$, 103</td>
<td></td>
</tr>
<tr>
<td>$E(u)$, 63</td>
<td></td>
</tr>
<tr>
<td>$E^*(u)$, 67</td>
<td></td>
</tr>
<tr>
<td>$E_n$, 103</td>
<td></td>
</tr>
<tr>
<td>$G + (x, S)$, 16</td>
<td></td>
</tr>
<tr>
<td>$L(q)$, 25</td>
<td></td>
</tr>
<tr>
<td>$M(u)$, 63</td>
<td></td>
</tr>
<tr>
<td>$N$, 103</td>
<td></td>
</tr>
<tr>
<td>$NE(u)$, 63</td>
<td></td>
</tr>
<tr>
<td>$N[x]$, 16</td>
<td></td>
</tr>
<tr>
<td>$P(u)$, 63</td>
<td></td>
</tr>
<tr>
<td>$P^*(u)$, 67</td>
<td></td>
</tr>
<tr>
<td>$T(S)$, 16</td>
<td></td>
</tr>
<tr>
<td>$\epsilon$, 16</td>
<td></td>
</tr>
<tr>
<td>$\sigma(u)$, 83</td>
<td></td>
</tr>
<tr>
<td>$\sigma[S]$, 30</td>
<td></td>
</tr>
<tr>
<td>$k_f$, 91</td>
<td></td>
</tr>
<tr>
<td>$k_i$, 91</td>
<td></td>
</tr>
<tr>
<td>$k_u$, 91</td>
<td></td>
</tr>
<tr>
<td>Accessibility graph, 24</td>
<td></td>
</tr>
<tr>
<td>Accessible, 23</td>
<td></td>
</tr>
<tr>
<td>Active edge, 35</td>
<td></td>
</tr>
<tr>
<td>Active list, 35</td>
<td></td>
</tr>
<tr>
<td>Alphabet, 16</td>
<td></td>
</tr>
<tr>
<td>Bookend, 118</td>
<td></td>
</tr>
<tr>
<td>Centre of a star, 16</td>
<td></td>
</tr>
<tr>
<td>Child-extremity node-join, 100</td>
<td></td>
</tr>
<tr>
<td>Chord diagram, 28</td>
<td></td>
</tr>
<tr>
<td>Circle graph, 28</td>
<td></td>
</tr>
<tr>
<td>Cleaned, 72</td>
<td></td>
</tr>
<tr>
<td>Cleaning, 72</td>
<td></td>
</tr>
<tr>
<td>Clique, 16</td>
<td></td>
</tr>
<tr>
<td>Clique-join, 26</td>
<td></td>
</tr>
<tr>
<td>Clique-split, 26</td>
<td></td>
</tr>
<tr>
<td>Co-component, 15</td>
<td></td>
</tr>
<tr>
<td>Coarser partition, 16</td>
<td></td>
</tr>
<tr>
<td>Completeness constraint, 133</td>
<td></td>
</tr>
<tr>
<td>Component, 15</td>
<td></td>
</tr>
<tr>
<td>Connected component, 15</td>
<td></td>
</tr>
<tr>
<td>Consecutive (in a chord diagram), 116</td>
<td></td>
</tr>
<tr>
<td>Consecutive constraint, 119</td>
<td></td>
</tr>
<tr>
<td>Consistency constraint, 133</td>
<td></td>
</tr>
<tr>
<td>Dead, 39</td>
<td></td>
</tr>
<tr>
<td>Degenerate</td>
<td></td>
</tr>
<tr>
<td>Module, 17</td>
<td></td>
</tr>
<tr>
<td>Split, 20</td>
<td></td>
</tr>
<tr>
<td>Double occurrence word, 28</td>
<td></td>
</tr>
<tr>
<td>Empty marker vertex, 63</td>
<td></td>
</tr>
<tr>
<td>Encloses, 116</td>
<td></td>
</tr>
<tr>
<td>End-vertex, 30</td>
<td></td>
</tr>
<tr>
<td>Extremity of a star, 16</td>
<td></td>
</tr>
</tbody>
</table>
Factorizing permutation, 19
Frontier, 20
Fully-mixed, 64
GLT, 21
Grafting GLTs, 23
Graph-labelled tree, 21
Induced GLT, 23
Inheriting
   Marker vertices, 79
   Nodes, 79
Internal tree edge, 16
Label-edge, 22
last, 85
LBFS, 30
Letters, 16
Lexicographic Breadth-First Search, 30
Marker vertex, 22
Markers, 22
Maximal module, 17
Maximal slice partition, 34
Maximal slice tree partition, 36
MD tree, 17
Mixed marker vertex, 63
Modular Decomposition, 17
Modular decomposition tree, 17
Module, 17
   Maximal, 17
   Parallel, 17
   Prime, 17
   Series, 17
   Strong, 17
   Type, 18
Neighbourhood join, 20
Node, 16
Node-join, 25
Node-split, 26
Non-trivial, 16
Offspring, 79
Opposite, 22
Ordered partition, 16
Overlap, 16
Parallel
   Graph, 18
   Module, 17
Parent-extremity node-join, 100
Partially complemented realizer, 133
Partitionable (realizer), 122
PCR, 133
Perfect marker vertex, 63
Pivot, 35
Primary stamp, 111
Prime
   Graph (Modular Decomposition), 18
   Graph (Split Decomposition), 20
   Module, 17
Realizer
   Chord diagram, 28
   Double occurrence word, 28
Reduced GLT, 26
Refinement, 38
Refinement of a partition, 16
Regular node-join, 100
Root marker vertex, 88
Secondary Stamp, 111
Series
  Graph, 18
  Module, 17
Slice, 31
Spine, 51
Split, 19
  Frontier, 20
  Operation, 20
Split Decomposition, 20
Split Decomposition Tree, 20
Split-tree, 27
Stamp
  Primary, 111
  Secondary, 111
Star, 16
Star-join, 26
Star-split, 26
String, 16
Strong Module, 17
Substring, 16
Tree partition, 36
Tree-edge, 22
Trivial, 16
Twins, 16
universal, 85
Zombie, 44